UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CURSO DE GRADUAÇÃO EM ENGENHARIA ELETRÔNICA

LUCAS FELIPE ROCKENBACH

SISTEMA QUADROTOR: UMA APLICAÇÃO DE FUSÃO DE SENSORES E FUNDAMENTOS DE CONTROLE

TRABALHO DE CONCLUSÃO DE CURSO

TOLEDO
2018

LUCAS FELIPE ROCKENBACH

SISTEMA QUADROTOR: UMA APLICAÇÃO DE FUSÃO DE SENSORES E FUNDAMENTOS DE CONTROLE

Trabalho de Conclusão de Curso, apresentado como requisito parcial para obtenção do título de Bacharel em Engenharia Eletrônica, na Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Ruahn Fuser

Co-orientadora: Prof. Ma. Jaqueline Vargas

TOLEDO

2018

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Toledo
Coordenação do Curso de Engenharia Eletrônica

TERMO DE APROVAÇÃO

Título do Trabalho de Conclusão de Curso N° _____

SISTEMA QUADROTOR: UMA APLICAÇÃO DE FUSÃO DE SENSORES E FUNDAMENTOS DE CONTROLE

por

Lucas Felipe Rockenbach

Este Trabalho de Conclusão de Curso foi apresentado às 10:00 h do dia 21 de novembro de 2018 como requisito parcial para obtenção do título **Bacharel em Engenharia Eletrônica**. Após deliberação da Banca Examinadora, composta pelos professores abaixo assinados, o trabalho foi considerado **APROVADO**.

_____
Prof. Me. Marcos Roberto Bombacini
(UTFPR-TD)

_____
Prof. Dr. Felipe Walter Dafico Pfrimer
(UTFPR-TD)

_____
Prof. Ruahn Fuser
(UTFPR-TD)
Orientador

Visto da Coordenação

_____
Prof. Me. Fábio Rizental Coutinho
Coordenador da COELE

O Termo de Aprovação assinado encontra-se na Coordenação do Curso

AGRADECIMENTOS

Agradeço a Deus pela vida. Aos meus Pais, Felipe Luis Rockenbach e Terezinha Maria Schardong Rockenbach, pelo amor e apoio em cada passo de minha vida, ao meu irmão Mateus Luís Rockenbach, que sempre tem estado comigo, me motivando. Aos meus professores orientadores, Ruahn Fuser e Jaqueline Vargas, pela amizade, a orientação e a confiança depositada. A Universidade Tecnológica Federal do Paraná, Campus de Toledo, especialmente ao Professor Jorge Dolores Vergara Dietrich, pela inicialização deste trabalho e ao professor Felipe Walter Dafico Pfrimer pelas ideias inspiradoras. Aos colegas Juliano da Rocha Queiroz e Jéssica dos Santos Hotz , pelo apoio e colaboração incondicional durante o desenvolvimento do trabalho e em especial a amizade de alguns amigos e colegas, que durante o curso de graduação fizeram parte de minha trajetória, Anderson Carlos Woss, Tyndalle do Santos Silva, Marcos Paulo Pettermann Bracht. A todos muito obrigado.

RESUMO

A utilização e estudo de Veículos Aéreos Não Tripulados (VANTs) aumentou consideravelmente nos últimos anos. Tais aeronaves encontram aplicações em ações de defesa e segurança, monitoramento de atividades agrícolas, mapeamento de regiões de difícil acesso, transporte de materiais, entre outras. É apresentado neste trabalho a categoria de quadrotores, descrevendo seus dispositivos e como são integrados ao sistema. Adicionalmente, são investigadas as características técnicas e a programação do código. Neste protótipo, o microcontrolador realiza a comunicação com sensores de aceleração, giro e campo magnético, além de gerenciar transferência de informações com o receptor de rádio. Por meio de algoritmos computacionais, é realizada a estimação da cinemática, determinando com precisão a posição angular nos eixos horizontais. A utilização desses métodos, tornou possível a implementação do controlador PID (proporcional-integral-derivativo) e trouxe como resultado final um quadrotor capaz de movimentar-se sob comandos de operação remota.

***Palavras-chave**: Quadrotor, VANT, Medidas Inerciais, Controle.*

ABSTRACT

The use and study of Unmanned Aerial Vehicles (UAV) have been increased considerably in the last few years. These aircrafts have applications on defense actions and security, monitoring agriculture activities, mapping zone of difficult access, transporting materials, among others. It is show in this work the quadrotor category, describing your devices and how they are integrated to the system. Additionally, technical characteristics and code programming are investigated. In this prototype, the microcontroller performs communication with acceleration, spin and magnetic field sensors, besides, it manages information transfer with the radio receptor. Through computational algorithms, it makes the kinematics estimation, defining accurate angular position of horizontal axes. These methods' use makes possible the implementation of PID (proportional-integrative-derivative) controller and brings as final result a quadcopter capable of self-motion by remote operation commands.

Keywords: *Quadrotor, UAV, Inertial measurement, Control.*

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE SIGLAS E ABREVIATURAS

AHRS	*Attitude and Heading Reference Systems*
ANATEL	Agência Nacional de Telecomunicações
API	*Application Programming Interface*
ARM	*Advanced RISC Machine*
DC	*Direct Current*
DCM	*Direction Cosine Matrix*
DECEA	Departamento de Controle do Espaço Aéreo
DoD	*United States Department of Defense*
EKF	*Extended Kalman Filter*
ESC	*Electronic Speed Controller*
FM	Frequência Modulada
I2C	*Inter-Integrated Circuit*
IMU	*Inertial Measurement Unit*
LCC	*Leadless Chip Carrier*
LGA	*Land Grid Array*
LiPo	*Lithium-ion Polymer*
LQR	*Linear-Quadratic Regulator*
LSb	*Least Significant bit*
MARG	*Magnetic, Angular Rate and Gravity*
MEMS	*Micro-Electro-Mechanical Systems*
MOSFET	*Metal Oxide Semiconductor Field Effect Transistor*
PID	Proporcional Integral Derivativo
PPM	*Pulse Position Modulation*
PWM	*Pulse Width Modulation*
RISC	*Reduced Instruction Set Computer*
RF	Rádio Frequência
RPA	*Remotely Piloted Aircraft*
SPI	*Serial Peripheral Interface*
STARMAC	*Stanford Test Bed of Autonomous Rotorcraft for Multi Agent Control*
UAV	*Unmanned Aircraft Vehicle*
VTOL	*Vertical Take-off and Landing*

SUMÁRIO

1 INTRODUÇÃO

Os multirotores são uma categoria de helicópteros com três ou mais rotores, que geralmente usam hélices de passo fixo, de modo que a movimentação é o resultado da relação de rotação entre os rotores [1]. Devido a sua versatilidade, vários centros de pesquisa e desenvolvimento ao redor do mundo focam em pesquisas relacionadas a esses dispositivos. As aplicações vão desde o mapeamento de terrenos até a entrega de produtos, podendo também ser utilizados em sistemas de vigilância [2].

O quadrotor (objeto em estudo neste trabalho), como o próprio nome sugere, é um tipo de helicóptero multirotor que é sustentado e impulsionado por quatro rotores. Tais dispositivos também são conhecidos como quadricópteros. A concepção dessa aeronave não é recente, existindo relatos de aparelhos construídos e testados desde o início do século XX. Os primeiros modelos eram tripulados, assim como o caso do Giroplano de Breguet-Richet [3], o Oehmichen No. 2 de Etienne Oehmichen e o helicóptero do Dr. George de Bothezat [4]. Atualmente, tanto os quadricópteros como outros tipos de multirotores são projetados e desenvolvidos para serem controlados remotamente ou operarem de forma autônoma, resultando em dispositivos leves, de pequena envergadura e de baixo custo.

O surgimento de multirotores não tripulados é resultado de diversos avanços da indústria eletrônica. Os dispositivos MEMS (*Micro-Electro-Mechanical Systems*) possibilitaram que os instrumentos de medição acelerômetros, giroscópios, magnetômetros, barômetros e ultrassons pudessem ser embarcados em multirotores, pois têm dimensões e pesos reduzidos. A simplicidade de construção dos motores de corrente continua sem escova, garantiu potências e torques significativos com baixo consumo de energia. Aliado a isso, a tecnologia das baterias avançou consideravelmente para maiores relações de carga-massa. O crescente uso destes dispositivos em bens de consumo culminou na produção em larga escala e a diminuição dos custos de aquisição.

Assim, com os diferentes avanços tecnológicos mencionados, este trabalho pretende apresentar a relação entre alguns elementos para constituir o quadrotor. As referências são passadas por rádio comunicação, os dados são adquiridos na Unidade de Medição Inercial (IMU, sigla inglesa para *Inertial Measurement Unit*) e o processamento dos filtros e controlador são executados na placa de desenvolvimento Tiva C^{TM}.

Em vista do amplo escopo de engenharia e pesquisa envolvidos neste equipamento, o presente estudo mantém em segundo plano a seleção das peças que constituem os VANTs, pois tal avaliação acarretaria em critérios complexos, desde aerodinâmicos até computacionais.

Neste trabalho é oferecida uma abordagem teórica e prática para enriquecer a noção de como diversos componentes e ferramentas interagem com a finalidade de estabelecer a estabilidade da posição angular do quadrotor.

1.1 OBJETIVOS

Objetivo Geral: Montar um protótipo de quadrotor para estabilização de seus eixos por meio de métodos de filtragem e controle clássico.

Objetivos Específicos:

Montar as peças da estrutura;

Interligar os dispositivos periféricos (rádio, sensores, atuadores);

Programar a comunicação entre os dispositivos;

Obter a posição inercial em tempo real;

Implementar técnica de controle clássico;

1.2 JUSTIFICATIVA

A tecnologia utilizada nos multirotores ainda é pouco conhecida e com custo elevado para utilização profissional. Nas últimas décadas tem sido estudada a dinâmica de voo e técnicas de controle para sua estabilização com o intuito de melhorar a resposta do sistema e reduzir o custo em montagem e energia. Associado a isso, ainda há dificuldades na filtragem e condicionamento dos sinais entregues pelos sensores, gerando complexidade na implementação de um sistema de controle clássico, fatores que devem ser minuciosamente estudados para o sucesso do projeto.

Além disso, a investigação das características detalhadas deste veículo aéreo não tripulado são fundamentais para a eficácia das aplicações, dentre elas pode-se citar: proteção civil (apoio a coordenação e comando no combate a incêndio, e operações de busca e salvamento), manutenção de estruturas (inspeção de estruturas, planejamento de obras em fábricas, manutenção de estradas), fotografia aérea e vídeo (cobertura jornalística de eventos, cinema, filmes publicitários, fotografia de animais selvagens, fotografia profissional), mapeamento aéreo (geoprocessamento de áreas), segurança (inspeção de zonas críticas, alterações de ordem pública, vigilância de perímetros), proteção ambiental (investigação de

acidentes ambientais), militar (vigilância militar tática e ataque) e aplicação da lei (investigação de cena de crime e recolhimento de informação, investigação de acidentes de viação, análise de congestionamento de trânsito, eliminação de engenhos explosivos) [5].

1.3 ORGANIZAÇÃO DO TEXTO

Os conceitos utilizados no trabalho surgem a partir do conhecimento sobre a temática, sendo assim a consulta à bibliografia referenciada é imperativa. O desenvolvimento textual é coerente ao cronograma prático, se iniciando em um estudo preliminar e visão geral do projeto e se encerrando com conclusões das implementações, conforme segue abaixo:

No capítulo 2 está contida a revisão teórica sobre os veículos aéreos não tripulados e multirotores no contexto científico. As características fundamentais de cada material utilizado no projeto são descritas no capítulo 3. A partir do capítulo 4 são apresentadas as etapas de execução do projeto bem como desafios encontrados. O capítulo 5 oferece um diagnóstico sobre as etapas e seus resultados. No capítulo 6 são apresentadas as referências bibliográficas. Por fim, o Apêndice A traz o código fonte desenvolvido neste projeto.

2 PLATAFORMA QUADROTOR

Neste capítulo será apresentado o contexto histórico e teórico do assunto, bem como, as soluções utilizadas no desenvolvimento da tecnologia.

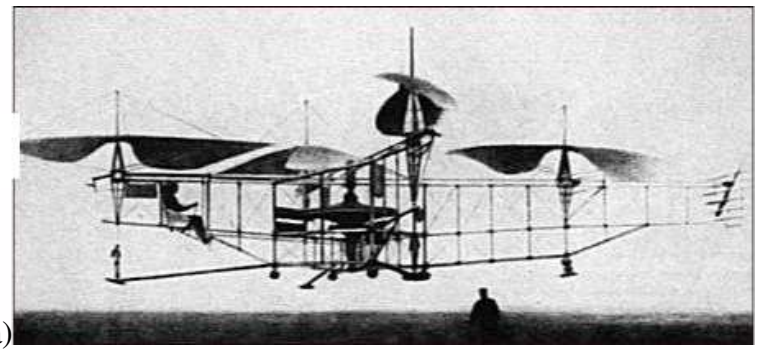
A capacidade de voar dos pássaros fascinou a humanidade durante toda sua história, o primeiro voo de um objeto mais pesado que o ar, foi idealizado pelo brasileiro Alberto Santos Dumont, o avião 14 Bis superou a gravidade utilizando propulsão própria e manteve-se no ar por 60 metros, em 23 de outubro de 1906, sob olhar de mil espectadores [6].

A primeira plataforma voadora com asas rotativas surgiu por volta de 1907, construída por Louis e Jacques Breguet, orientados pelo Professor Charles Richet, levantou a poucos metros do solo com o auxílio de muitos homens [3].

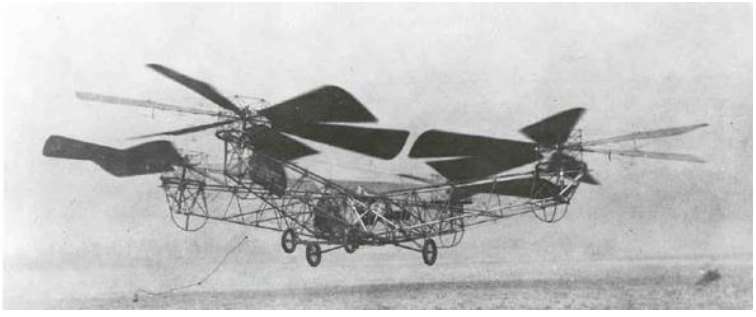
Em 1920, o inventor Étienne Oehmichen criou a aeronave Oehmichem No. 2, mostrada na Figura 1a. Possuía estabilidade e controlabilidade razoáveis para a época, porém exigia muito trabalho do piloto para realizar movimentos. Outra máquina voadora que ficou para a história foi desenvolvida por George de Bothezat e Ivan Jerome, mostrada na Figura 1b. Atingiu a marca de cinco metros de altura, entretanto possuía baixa potência e acumulava problemas de confiabilidade [4].

Figura 1 - a) Quadrotor Oehmichem em 1922; b) Bothezat em 1923

a)



b)



Fonte: a) Domínio público [7]; b) Thomas Edison National Historical Park [8]

Milhares de voos foram realizados a fim de tornar os quadricópteros uma opção de transporte, mas a tecnologia do século XX não era suficiente, deixando esta aeronave esquecida por décadas. Somente a partir dos anos 2000 aeromodelos na forma de quadrotores passaram a ser empregados, em consequência dos avanços tecnológicos de sensores, motores elétricos, baterias e microprocessadores. Nesse período, a pilotagem tornou-se remota, sendo os comandos enviados por comunicadores de rádio frequência, e auxiliada por equipamentos eletrônicos para realizar o controle do voo.

Dentre os controladores de voo mais populares, tem-se as placas APM^{TM}, PIX4D^{TM} e PIXHAWK^{TM}, projetados desde o princípio para executar o código aberto Ardupilot sobre a plataforma de desenvolvimento Arduino^{TM}. Outra placa resultado da união de desenvolvedores independentes é a Multwii, elaborada a partir de componentes eletrônicos fabricados em larga escala e custo reduzido. Por fim, as placas Naza M^{TM} e A2^{TM} produzidas pela DJI, possuem grande aceitação no mercado de consumo, por conta da praticidade e versatilidade.

O fundador da Amazon, Jeff Bezos, anunciou em 2014, um projeto para viabilizar a utilização de veículos aéreos não tripulados na entrega de pequenos pacotes. A agilidade e o impacto ambiental reduzido são as justificativas para implementar esse serviço [9]. A Figura 2 traz o protótipo que a Amazon vem aperfeiçoando com a pretensão de utilizar em entregas rápidas, estima-se a realização de entregas em trinta minutos para locais a dezesseis quilômetros do depósito de produtos.

Figura 2 - Protótipo Amazon Prime Air

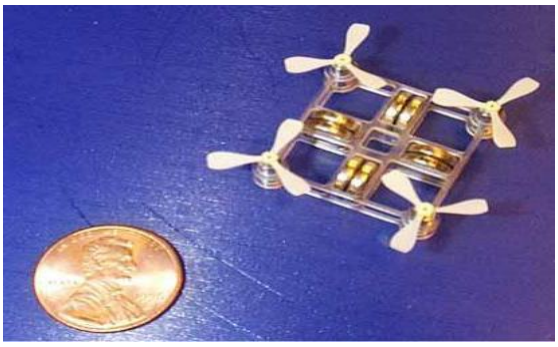
Fonte: Amazon.com, Inc. [9]

2.1 PROJETOS ACADÊMICOS

Como o quadricóptero apresenta um problema de controle relativamente complexo, diversos estudos foram iniciados em universidades. Devido as suas vantagens características e o potencial de aplicação, expressivos investimentos públicos e privados estão sendo feitos, com vista a aperfeiçoar a tecnologia de voo nas mais variadas condições de ambientes.

A Universidade de Stanford (*Stanford University*) iniciou em 1999 o projeto Mesicopter, mostrado na Figura 3 com o objetivo de avaliar a possibilidade de desenvolver um quadrotor na escala de um centímetro. A intenção dos pesquisadores era utilizar esses modelos em grande quantidade para estudar a situação climática do planeta [10].

Figura 3 - Mesicopter



Fonte: Universidade de Stanford [10]

Em 2005 surgiu a plataforma STARMAC (*Stanford Test Bed of Autonomous Rotorcraft for Multi Agent Control*), realizada na Universidade de Stanford, com a finalidade de testar os novos conceitos de controle multi-agente em ambientes abertos. Esse modelo pode executar tarefas complexas, pois foi equipado com instrumentos de medida precisos e alto poder de computação. Foi utilizada a teoria do controle ótimo LQR (*Linear-Quadratic Regulator*) para a estabilização e a técnica *Sliding Mode* para a altitude [11].

Na tese de doutorado de Samir Bouabdallah [12], foram avaliados os problemas envolvendo quadrotores e as possiveis soluções por métodos de controle. Nos laboratórios da Escola Federal Politécnica de Lausanne (*Ecole Polytechnique Fédérale de Lausanne*), foram utilizadas câmeras para estimar a posição e aplicar diversas técnicas, entre elas, PID, LQR, *Sliding Mode* e *Backstepping* [13] [14]. Verificou-se que a técnica de controle por modos deslizantes introduz vibrações de alta frequência e baixa amplitude devido ao seu comportamento de comutação, enquanto que o controlador obtido com a técnica *backstepping* provou sua robustez ao rejeitar grandes perturbações nos eixos do quadrotor [15].

2.2 CARACTERÍSTICAS DOS QUADROTORES

Os veículos aéreos não tripulados (VANT) são assim definidos por não possuírem uma cabine de pilotagem. São guiados por controle remoto, sem fio e a distância, em modo automático com supervisão de um operador ou de maneira autônoma. Na definição do DoD (*United States Department of Defense*) eles são conhecidos como UAV (*Unmanned Aerial Vehicles*) [16]. Com os mesmos conceitos, no Brasil, o Departamento de Controle do Espaço Aéreo (DECEA) define as Aeronaves Remotamente Pilotadas (RPA, sigla inglesa para *Remotely Piloted Aircraft*) [17]. Inserido nas terminologias apresentadas, o termo mais conhecido popularmente é drone, possuindo uma definição genérica.

Dentre os VANTs, os multirotores possuem a habilidade de decolar e pousar verticalmente e por isso são chamados de VTOL (*Vertical Take-off and Landing*). Sendo também capazes de realizar um voo pairado, mover-se para frente, para os lados e circular no próprio eixo com precisão e manutenção da altitude. Em outras palavras, são capazes de mudar completamente sua direção de voo com agilidade e, da mesma forma, deter seu movimento abruptamente [18][19].

O quadrotor ou quadricóptero é um multirotor, de configuração simples, constituído por quatro rotores fixos instalados em cada um dos quatro cantos de uma estrutura cruzada, na forma da Figura 4, que embarca no seu centro o conjunto de equipamentos necessários para o funcionamento.

Figura 4 - Aparência típica de um quadricóptero

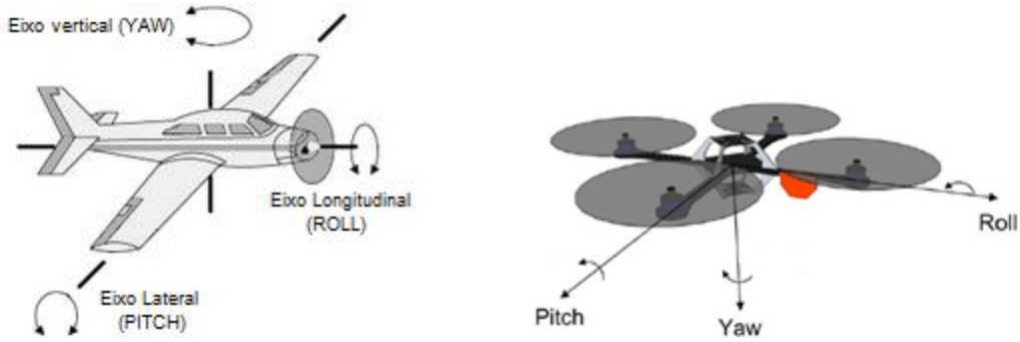
Fonte: Aeryon Labs [20]

Sua maneira de movimentar-se no espaço é através da alteração de inclinação na estrutura. Tal condição é obtida através da diferença de rotação entre os motores, ou seja, caso não exista inclinação dos eixos de orientação do conjunto, o quadrotor permanece imóvel

levitando. Para um voo estável, deve haver equilíbrio horizontal sobre o eixo central. Cada conjunto de motor e hélice executa essa tarefa ao aplicar empuxo na posição que estão alocadas, equalizando a força gravitacional presente.

O ponto de origem da orientação de uma aeronave é seu centro de massa e, de maneira geral, a rotação dos seus eixos (*axis*) determina a direção do movimento. Os eixos de orientação na aviação são denominados arfagem (*pitch)*, rolamento (*roll)* e guinada (*yaw)*, como pode ser observado na Figura 5.

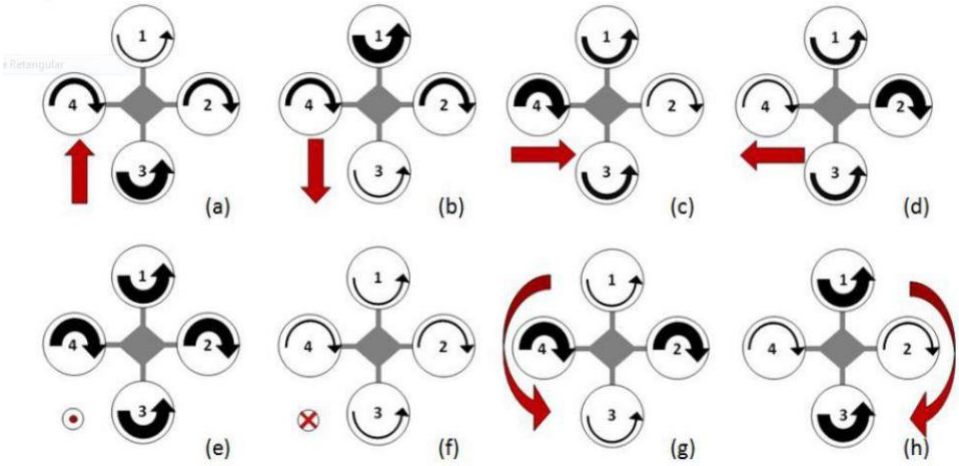
Figura 5 - Orientação na aviação e orientação do quadrotor



Fonte: Licença Publica Geral (GPL) [21] e Exuperry B. Costa [22]

Na Figura 6 é possível observar as diferenças de velocidade entre os motores, necessárias para produzir a movimentação do quadrotor. As setas espessas indicam maiores velocidades, as setas estreitas menores velocidades e as setas médias representam uma condição neutra.

Figura 6 - Movimentação do quadrotor de acordo com as velocidades dos motores: (a) e (b) arfagem, (c) e (d) rolamento, (e) e (f) altitude, (g) e (h) guinada



Fonte: Rejane C. Sá [23]

Os motores colocados na direção transversal giram no sentido contrário à dos motores colocados na direção longitudinal. Essa disposição anula o torque produzido na base do motor com direção contrária ao sentido de rotação. A situação de equilíbrio pode ser alterada para provocar o movimento de guinada, feito em torno do eixo vertical do quadricóptero, para isso os dois motores girando no sentido horário devem estar com velocidades diferentes em relação aos dois motores girando no sentido anti-horário, conforme representado nas Figuras 6(g) e 6(h).

No sentido do eixo longitudinal do centro de massa do aeromodelo o movimento é provocado através da diferença de velocidade entre o motor da frente e o de trás. O movimento de arfagem acontece quando o empuxo gerado pela diferença de rotação das hélices inclina o quadricóptero, levando a mudança de posição [24], como demonstrado nas Figuras 6(a) e 6(b).

A movimentação no eixo transversal ocorre quando um dos motores laterais é acelerado e o outro desacelerado, como está ilustrado nas Figuras 6(c) e 6(d), fazendo com que o quadricóptero incline para o lado contrário. Esse giro em torno do ponto central é chamado rolamento.

Para o quadricóptero manter posição estacionária, de total equilíbrio, em um ponto do espaço tridimensional mais um fator deve ser analisado. A manutenção da altitude em relação ao solo é obtida quando a força de empuxo produzida pelos quatro rotores estiver igualada ao módulo da força-peso gerada pela aceleração gravitacional. O movimento vertical de subida e descida, é feito aumentando ou diminuindo a velocidade de todos os motores igualmente, representado nas Figuras 6(e) e 6(f).

3 MATERIAIS

Neste capítulo serão apresentados os componentes elementares que constituem o quadricóptero, como também os meios de interação dos componentes para trocar as informações.

3.1 ARMAÇÃO

O quadricóptero caracteriza-se na forma de uma aeronave com quatro braços simétricos, formando uma cruz, como pode ser visto na Figura 7 sendo que em cada extremidade é fixado um conjunto de motor com hélice. No centro são alocados os equipamentos necessários para o funcionamento, em especial uma bateria, componente com o maior peso.

Figura 7 - Armação de quadrotor

Fonte: HobbyKing [25]

O grande requisito relacionado à armação é a rigidez da estrutura, pois assim os instrumentos de medição obtêm dados concisos sobre a orientação do aeromodelo.

3.2 BATERIA

Há uma grande demanda de energia para sustentação do quadricóptero e o peso dos equipamentos que o compõem deve ser mínimo. Assim, a bateria precisa aliar baixo peso, grande capacidade de carga, e ainda, alta corrente imediata de descarga. Para esta aplicação o melhor desempenho é obtido com uma bateria de Polímero de Lítio (LiPo, sigla inglesa para

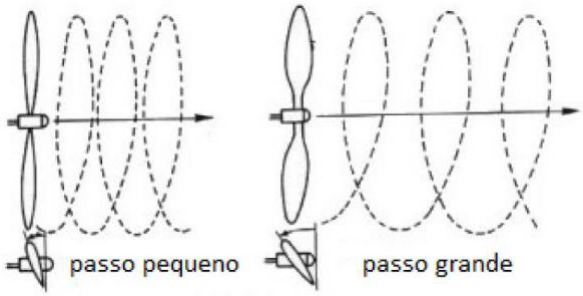
Lithium-ion Polymer), onde a tensão de saída é resultado da soma das tensões de cada célula. Utiliza-se neste projeto uma bateria com três células de 3,7 V, resultando em 11,1 V de tensão nominal. A capacidade de carga da bateria é 4000 mAh em virtude de sua relação carga-massa ser a mais adequada para quadrotores de médio porte.

Como em qualquer aeromodelo, a interrupção da alimentação em pleno voo é indesejável pois acarreta em queda e grandes danos. Para supervisionar a descarga da bateria é utilizado um dispositivo eletrônico que emite sinais sonoros e luminosos quando a tensão mínima utilizável é medida. Outra função desses sinais é alertar o usuário para substituir a bateria ou carregá-la para não sofrer danos na estrutura química do polímero de lítio.

3.3 HÉLICE

A hélice é um componente geralmente feito de nylon, fibras de vidro ou fibras de carbono. Hélices de maior comprimento possuem eficiência resultante maximizada, entretanto as limitações do conjunto motor e ESC devem ser respeitadas. Outra característica fundamental é o passo da hélice, relativo a inclinação das pás, que determina o avanço obtido por uma rotação completa da hélice, como está ilustrado na Figura 8.

Figura 8 - Passo da hélice



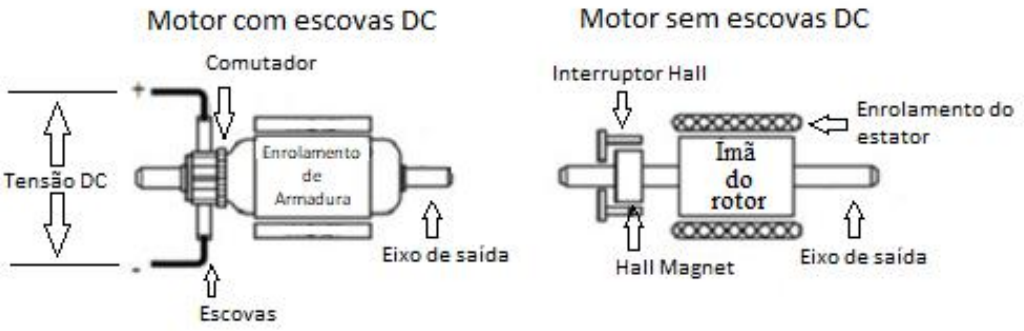
Fonte: Exuperry B. Costa [22]

3.4 MOTOR

Os elementos finais de controle são quatro motores de corrente contínua sem escovas, mais conhecidos por *Brushless* DC (*Direct Current*). O motor de escovas possui uma armadura que atua como um eletroímã, com dois polos, e funciona invertendo o sentido da corrente quando as escovas entram em contato com o comutador [5], assim como está representado na Figura 9 (a). No motor sem escovas a comutação é realizada eletronicamente, para isso é

necessário detectar a posição do rotor e aplicar tensão no próximo enrolamento para manter a rotação [26], como está representado na Figura 9 (b).

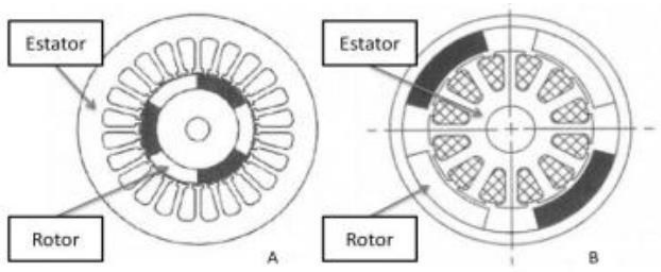
Figura 9 - Esquema de comutação das bobinas para: (a) motor DC com escovas e (b) motor DC sem escovas



Fonte: J. C. S. VIEIRA [26]

Entre as vantagens dos motores *Brushless* estão pouca necessidade de manutenção, tempo de vida útil estendida, menor ruído sonoro, tamanho e peso reduzidos, e ainda, menor produção de interferência eletromagnética. Tais características acarretam em um custo maior quando comparado ao motor DC tradicional, além de precisar de um circuito eletrônico de controle. Há duas variações na construção do motor *Brushless*, o tipo *outrunner* e *inrunner*. A maioria dos aeromodelos utiliza o *outrunner,* em que o elemento móvel de rotação é o estator, onde são montados os imãs permanentes. No suporte de centro ficam localizadas as bobinas, assim como pode ser observado na Figura 10 (a). As propriedades do motor *outrunner* produzem maior torque e menor velocidade quando comparado ao tipo *inrunner*, na qual utiliza ímãs permanentes fixos no eixo do motor e as bobinas montadas no estator, assim como pode ser observado na Figura 10 (b).

Figura 10 - Construção de motores *Brushless DC*: (a) *inrunner*, (b) *outrunner*



Fonte: J. C. S. VIEIRA [26]

3.5 CONTROLADOR ELETRÔNICO DE VELOCIDADE (ESC)

Para cada motor *Brushless* do quadrotor deve haver um ESC alimentado diretamente pela bateria. Esse dispositivo deve possuir como entrada um sinal de Modulação por Largura de Pulso (PWM, sigla inglesa para *Pulse Width Modulation*), que é fornecido pela placa microcontrolada.

A saída é formada por três ondas retangulares defasadas de 120° necessárias para acionar os motores. A variação da razão cíclica (*duty cycle*) do sinal PWM faz com que o ângulo da tensão de saída seja chaveado para controlar a velocidade e a potência de cada motor. A construção das ondas retangulares é feita através de tiristores do tipo MOSFET (*Metal Oxide Semiconductor Field Effect Transistor*) na configuração *push-pull*. Um microcontrolador interno faz o chaveamento do gatilho com o auxílio da leitura da força contra-eletromotriz nas armaduras do motor, obtendo assim a posição dos imãs permanentes.

3.6 SENSORES

Segundo definição de Madgwick [27], uma Unidade de Medidas Inerciais (IMU, sigla inglesa para *Inertial Measurement Unit*) possui os sensores acelerômetro e giroscópio, sendo ela capaz de mensurar a atitude de um corpo rígido. Uma MARG (*Magnetic, Angular Rate and Gravity*) é uma placa de sensores híbrida pois combina uma IMU e um magnetômetro de maneira a formar um Sistema de Atitude e Direção Relativa (AHRS, sigla inglesa para *Attitude and Heading Reference System*) com a capacidade de prover orientação relativa a força da gravidade e ao campo magnético.

No presente trabalho, utiliza-se a placa de sensores GY-80 composta por um acelerômetro, um giroscópio e um magnetômetro. As informações coletadas pelos sensores são transmitidas para a placa microcontrolada através do protocolo de comunicação digital I^2C (*Inter-Integrated Circuit*). O barramento duplo que constitui este modo de comunicação conecta todos os sensores paralelamente.

3.6.1 Acelerômetro

O acelerômetro tem a capacidade de medir acelerações estáticas, como a gravidade, e acelerações dinâmicas, causadas pela variação da velocidade do corpo. Esse sensor é capaz de mensurar componentes vetoriais paralelas ao campo gravitacional da Terra, logo ele é incapaz

de medir rotações paralelas ao plano terrestre. Portanto, o acelerômetro pode ser utilizado para mensurar os ângulos de arfagem e rolamento, mas não é capaz de mensurar o ângulo de guinada, pois este representa uma rotação em relação ao eixo paralelo ao plano terrestre. [28].

O acelerômetro utilizado nesse estudo foi o ADXL345 produzido pela companhia Honeywell. É um dispositivo leve, pequeno, e resistente, com 30 mg, dimensões iguais a 3 mm x 5 mm x 1 mm e capaz de resistir a impactos na ordem de 10000 unidades gravitacionais. Seu encapsulamento é de plástico do tipo LGA (*Land Grid Array*) com 14 pinos. A alimentação pode variar de 2,0 a 3,6 V e a faixa de temperatura suportada é -40 a 85 °C. A faixa de medição pode ser configurada em ±2 g, ±4 g, ±8 g e ±16 g; cada uma com a resolução de 10 bits, 11 bits, 12 bits e 13 bits, respectivamente; representado uma sensibilidade de 256 LSb/mg, 128 LSb/mg, 64 LSb/mg, 32 LSb/mg, respectivamente. O desvio de sensibilidade do ideal é ±1 %, o desvio de saída do ideal é ±35 mg e o erro de não linearidade ±0,5 %. [29]

3.6.2 Giroscópio

O giroscópio baseia-se na lei da inércia, princípio físico formulado por Newton, na qual um corpo em movimento mantém-se em movimento até que uma força externa seja aplicada sobre ele. Mais especificamente, os giroscópios de tecnologia MEMS detectam a aceleração de Coriolis nas peculiaridades dos movimentos oscilatórios [30]. A partir disso é possível obter a velocidade angular nos três eixos de orientação.

O giroscópio L3G4200D, utilizado nesse projeto, é um sensor inercial fabricado sobre pastilhas de silício em um processo desenvolvido pela companhia STMicroelectronics. Possui 16 pinos dispostos sob o encapsulamento LGA, com dimensões de 4 mm x 4 mm x 1,1 mm. A fonte de alimentação pode estar na faixa de 2,4 a 3,6 V, com capacidade de operar com temperaturas entre -40 a 85 °C. A faixa de medição pode ser configurada em ±250 °/s, ±500 °/s e ±2000 °/s; representado uma sensibilidade de 8,75 °/s/LSb, 17,5 °/s/LSb e 70 °/s/LSb, respectivamente; todos com resolução completa de 16 bits. O nível de desvio de taxa zero para cada faixa é ±10 °/s, ±15 °/s e ±75 °/s e o erro de não linearidade ±0,2 %. [31]

3.6.3 Magnetômetro

O magnetômetro é um dispositivo eletrônico com função semelhante a bússola tradicional. A finalidade é quantificar a posição em relação ao polo norte magnético e consequentemente obter o ângulo de guinada.

O dispositivo utilizado nesse estudo é um magnetômetro Honeywell HMC5883L, com 16 pinos montados em superfície LCC (*Leadless Chip Carrier*). Suas dimensões são 3 mm x 3 mm x 0,9 mm, peso de 18 mg, capacidade de suportar temperaturas entre -30 a 85 °C e alimentação na faixa 2,16 a 3,6 V. A faixa de medição pode ser configurada entre ±0,88 Ga e ±8,1 Ga; representado uma sensibilidade de 0,73 mG/LSb a 4,35 mG/LSb; todos com resolução de 12 bits. A tolerância ao ganho é ±5 % e o erro de não linearidade ±0,1 %.[32]

3.7 EMISSOR E RECEPTOR DE RÁDIO FREQUÊNCIA

O quadricóptero é comandado remotamente por um operador, utilizando um *link* de rádio frequência (RF) composto por um transmissor e um receptor, cuja portadora pode ser de 72 MHz ou 2,4 GHz em Frequência Modulada (FM) para esta finalidade. Tais frequências são regulamentadas no Brasil pela Agência Nacional de Telecomunicações (ANATEL).

Os comandos são modificados através de duas alavancas no rádio transmissor tradicional. Não existe uma definição para a disposição das alavancas, mas em geral, uma delas deve comandar a aceleração e a guinada, enquanto outra comanda a arfagem e o rolamento. São necessários, desta forma, no mínimo quatro canais de transmissão para estar enviando simultaneamente ao aeromodelo novas referências de comportamento em voo.

3.8 PLACA MICROCONTROLADA

A correta escolha da placa microcontrolada é fundamental para alcançar os resultados esperados. Atualmente os microcontroladores de 32 bits com arquitetura ARM (*Advanced RISC Machine*) destacam-se no mercado em diversas aplicações por conta de sua versatilidade e recursos. Dentro desta gama, a fabricante de componentes eletrônicos Texas Instruments oferece a placa de desenvolvimento TIVA^{TM} C, embarcada com o microcontrolador TM4C123GH6PMI.

Seu *chip* é construído com base em uma unidade de processamento ARM^{TM} Cortex-M4F de 32 bits e 80 MHz. A capacidade de armazenamento é 32 KB de memória RAM (*Random-Access Memory*) e 256 KB de memória *flash*. Neste trabalho são utilizadas 15 portas de entrada ou saída, 1 módulo UART, 1 módulo I^2C, 6 temporizadores de 64 bits e 1 temporizador de 32 bits [33] [34].

Os conteúdos didáticos apresentados em [35] e [36] deram suporte ao desenvolvimento da programação, como também a documentação oficial disponibilizada em [37] [38].

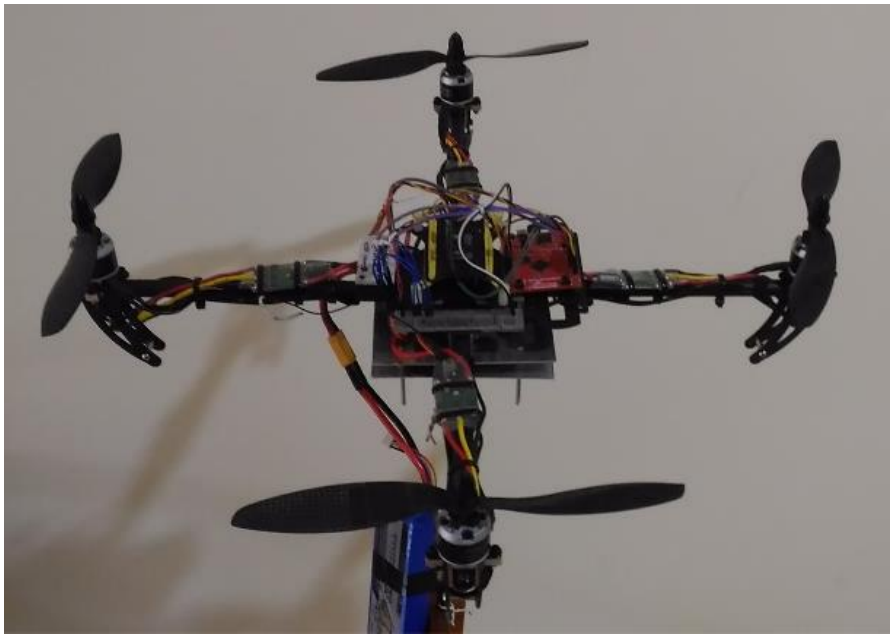
4 MÉTODOS E RESULTADOS

No decorrer deste capítulo será apresentado todo o processo de concepção do protótipo. Os conceitos e técnicas são aplicados diretamente no modelo real para analisar os resultados. Durante o desenvolvimento foram alterados diversas vezes os planos de execução, buscando sempre as abordagens mais objetivas. Todavia, neste relatório presou-se pela organização metodológica para obter didatismo na apresentação.

4.1 MONTAGEM E FUNCIONAMENTO

A disposição dos componentes do quadricóptero é importante para seu equilíbrio. Aproximar as peças do centro geométrico do aeromodelo possibilita ao sistema de controle melhor performance. Manter todas as partes devidamente presas é considerado imprescindível, pois assim vibrações são evitadas, como pode ser visto na Figura 11.

Figura 11 – Montagem do quadrotor

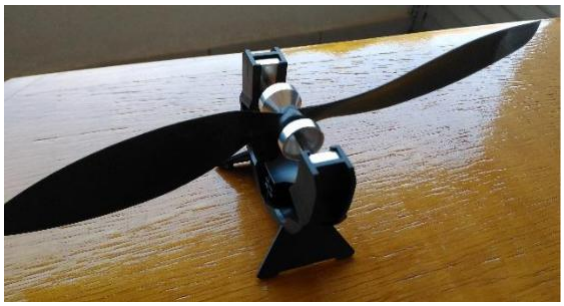


Fonte: Autoria Própria

As hélices que giram no sentido horário devem ser montadas na mesma diagonal, ficando em lados opostos. Já as hélices do sentido anti-horário de rotação precisam ser montadas na outra diagonal, estabelecendo equilíbrio entre as forças produzidas no eixo vertical.

Outra questão é o balanceamento mecânico, pequenas diferenças de peso entre os dois lados das hélices podem gerar grandes vibrações, visto que a rotação pode ultrapassar 10000 rpm. Para corrigir o desequilíbrio existente é utilizado um pequeno instrumento mecânico de rotação, como pode ser visto na Figura 12. De modo semelhante a uma balança, ele permite que a peça seja colocada em posição neutra e estável. Para corrigir as diferenças de peso são colocadas fitas adesivas no interior das hélices até obter o mesmo peso entre os dois lados.

Figura 12 - Instrumento de balanceamento de hélices

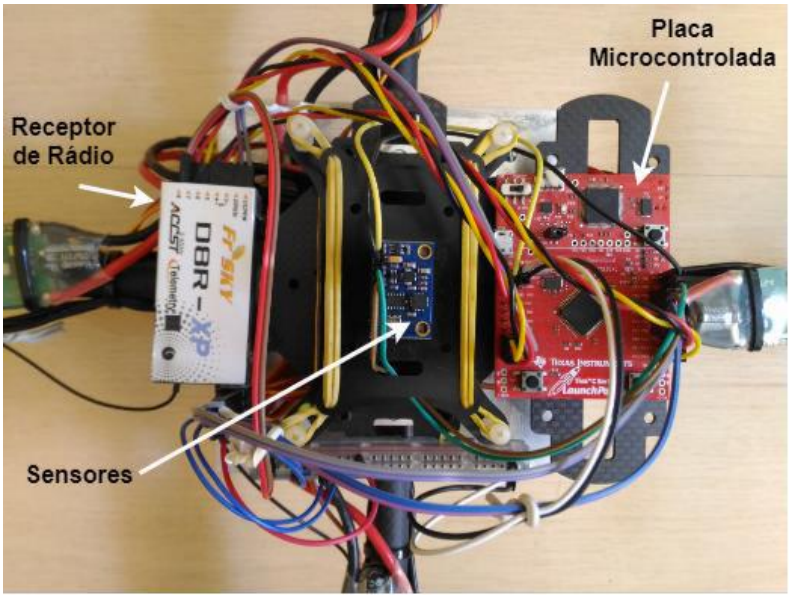


Fonte: Autoria Própria

A maior prioridade na montagem é dos sensores de medida inercial. Sua posição ideal é no centro de massa do conjunto montado, ou o mais próximo possível, como pode ser observado na Figura 13. É utilizado, adicionalmente neste protótipo, um acessório para absorção de vibrações. Trata-se de uma pequena peça de plástico suspensa nas extremidades por borrachas elásticas. Com isso os sensores acompanham o movimento da armação, mas sem receber oscilações mecânicas de alta frequência.

No quadrotor toda a energia é fornecida por uma bateria de 11,1 V em tensão contínua. Os seus terminais são conectados aos 4 ESCs diretamente. Nos ESCs um circuito chaveado converte a tensão contínua para um sinal PWM de 11,1 V que alimenta os 4 motores *Brushless* DC. Também existe um circuito complementar que fornece 5 V em tensão contínua, sendo este utilizado para alimentar a placa microcontrolada e o receptor de rádio. Na placa microcontrolada há um regulador de tensão para o nível de 3,3 V, adequado para alimentar a placa de sensores.

Figura 13 - Disposição dos componentes no centro da armação



Fonte: Autoria Própria

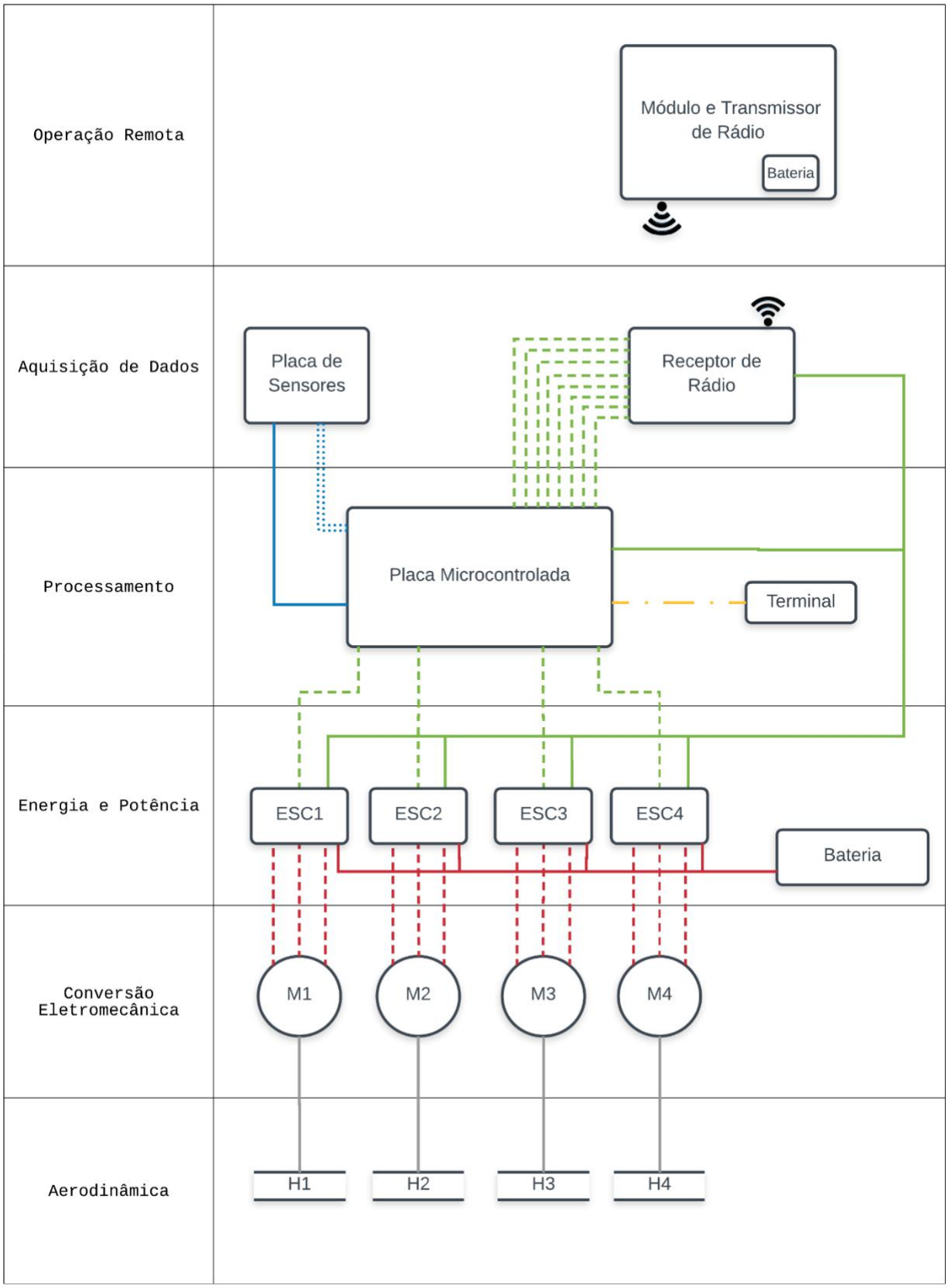
As informações são centralizadas na placa microcontrolada e, portanto, ela comunica-se com diversos dispositivos do quadrotor. Os dados dos sensores inerciais são coletados através de um barramento de comunicação I^2C de 3,3 V. As informações do receptor de rádio são capturadas por meio de 8 conexões PWM de 5 V. A rotação dos motores é definida por sinais enviados aos ESCs de cada um dos motores por meio de sinais PWM de 5 V. Assim, as hélices acopladas aos motores agem sobre a cinemática do aeromodelo. O Quadro 1 resume as conexões existentes e o diagrama da Figura 14 oferece uma representação esquemática da montagem do sistema.

Quadro 1 - Conexões do quadrotor

Origem	Destino	Conexão
Bateria	ESCs	11,1 V - CC
ESCs	Motores	11,1 V - PWM
ESCs	Placa microcontroladora	5 V-CC
ESCs	Rádio receptor	5 V-CC
Rádio receptor	Placa microcontrolada	5 V-PWM
Placa microcontrolada	ESCs	5 V-PWM
Placa microcontrolada	Placa de sensores	3,3V-CC
Placa de sensores	Placa microcontrolada	3,3 V - I^2C
Placa microcontrolada	Terminal	UART
Rádio transmissor	Rádio receptor	Wireless 2,4 GHz

Fonte: Autoria própria

Figura 14 - Diagrama de ligação



Fonte: Autoria Própria

Figura 15 - Legenda do diagrama de ligação

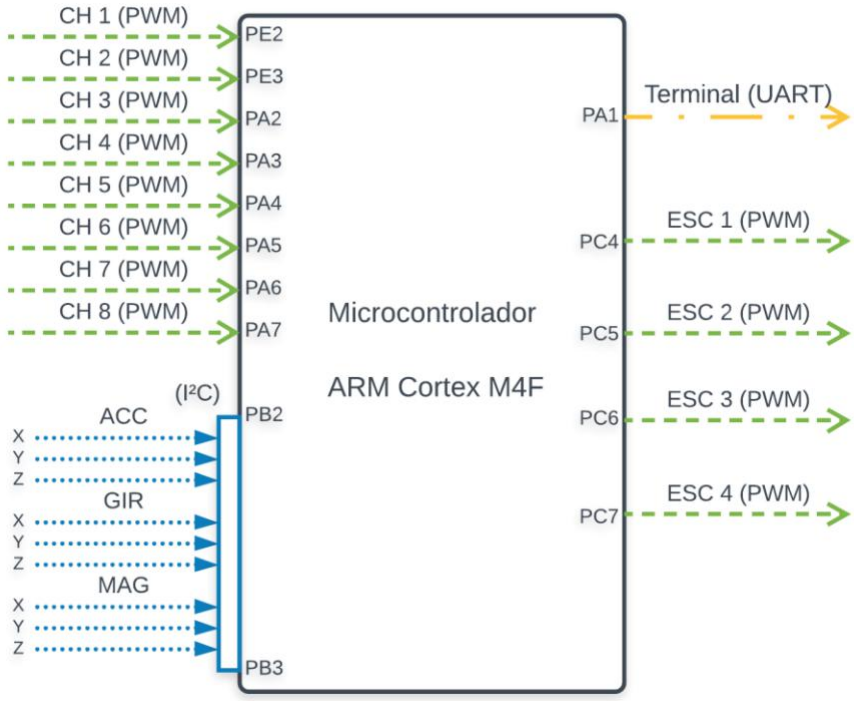
Fonte: Autoria Própria

4.2 VISÃO GERAL DO SISTEMA

A placa microcontrolada concentra grande quantidade de conexões, pois é ela quem de recebe, processa, armazena e envia informações aos demais dispositivos. Suas ações são executadas de maneira programada através de código escrito na linguagem C.

Os dispositivos periféricos que fornecem informações ao microcontrolador são vistos como entradas do sistema. Os comandos de orientação que movimentam o quadrotor, providos pelo controle remoto, são fornecidos através de 8 sinais PWM. Já as medidas de posição inercial, obtidas por meio do módulo de sensores GY-80, são transmitidas através de um barramento I^2C e compõem 9 valores de 16 bits. Por outro lado, os dispositivos que recebem informações do microcontrolador são saídas do sistema. A rotação dos motores é controlada através de 4 sinais PWM e os dados para depuração são enviados para o computador através de comunicação UART. A composição do sistema formado pelo microcontrolador ARM Cortex M4F com suas entradas e saídas está representado no diagrama da Figura 16.

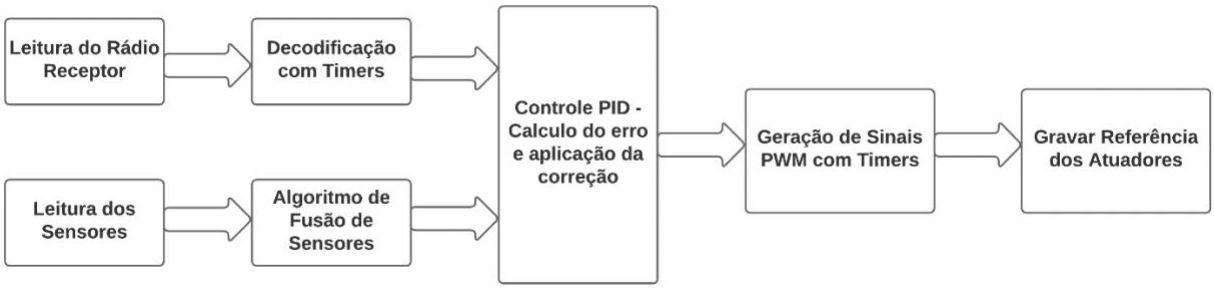
Figura 16 - Diagrama de entradas e saídas



Fonte: Autoria própria

O entendimento das ações do microcontrolador são simplificadas quando estão separadas em função das entradas e saídas existentes. As entradas dos canais de rádio, com sinal no formato PWM, são lidas e decodificadas da mesma forma. As entradas dos sensores inerciais passam pelo mesmo procedimento de leitura, são depois caracterizados e finalmente estimados os valores de posição inercial. No algoritmo de controle tais dados são postos em conjunto para estabelecer a resposta de reação. A atuação é realizada através de sinais no formato PWM, configurados para serem interpretados pelos atuadores. Deste modo, a Figura 17 descreve as funções internas do algoritmo do microcontrolador.

Figura 17 - Diagrama funcional do algoritmo

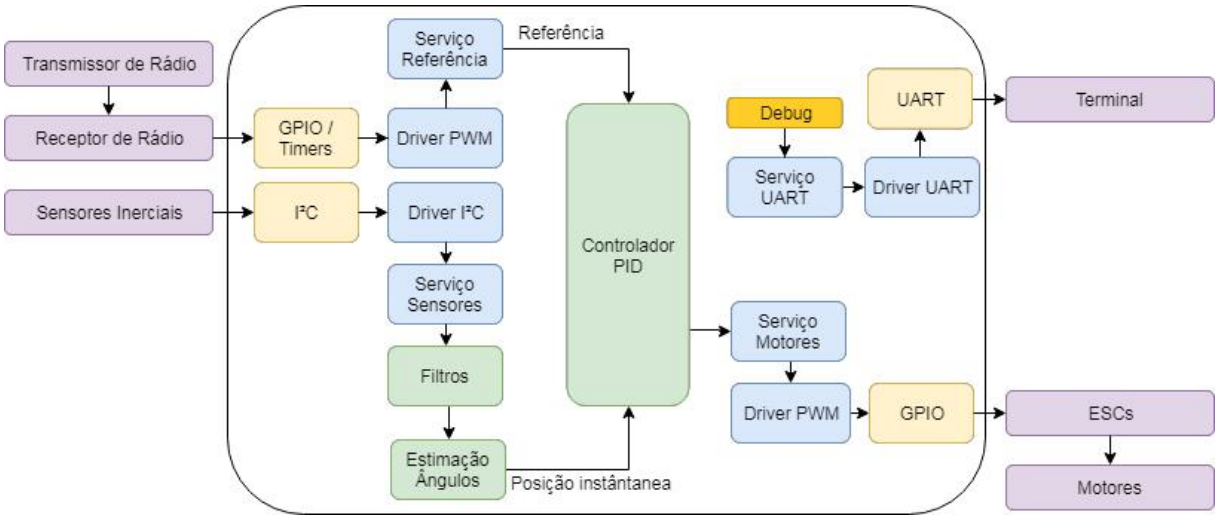


Fonte: Adaptado de Socialledge, Projeto S14 Quadcopte [39]

Apresentados os elementos externos e internos, todos para cumprir com o objetivo específico deste sistema embarcado, os elementos similares são agrupados. Os blocos formados elucidam uma sequência de etapas, porém, em diferentes níveis.

Nas bordas do microcontrolador os periféricos fazem a ponte de ligação com dispositivos externos. Os drivers utilizam comandos pré-definidos, chamados de API (*Application Programming Interface*), para leitura e escrita dos periféricos. Já os serviços, concluem a adaptação dos dados brutos para serem utilizados na filtragem, estimação e controle. Os dados de atuação são passados pelo controlador ao nível de serviço até chegar aos motores. O bloco de debug é capaz de transmitir ao terminal os dados relativos as demais etapas, a fim de coletar informações sobre o desempenho do sistema, conforme está representado na Figura 18.

Figura 18 – Arquitetura de firmware



Fonte: Autoria Própria

Com o funcionamento do microcontrolador apresentado, pode-se avançar na representação mais fiel do código. As instruções de execução estão divididas em grupos com objetivos definidos, chamados neste trabalho de tarefas, como pode ser visto na Figura 19.

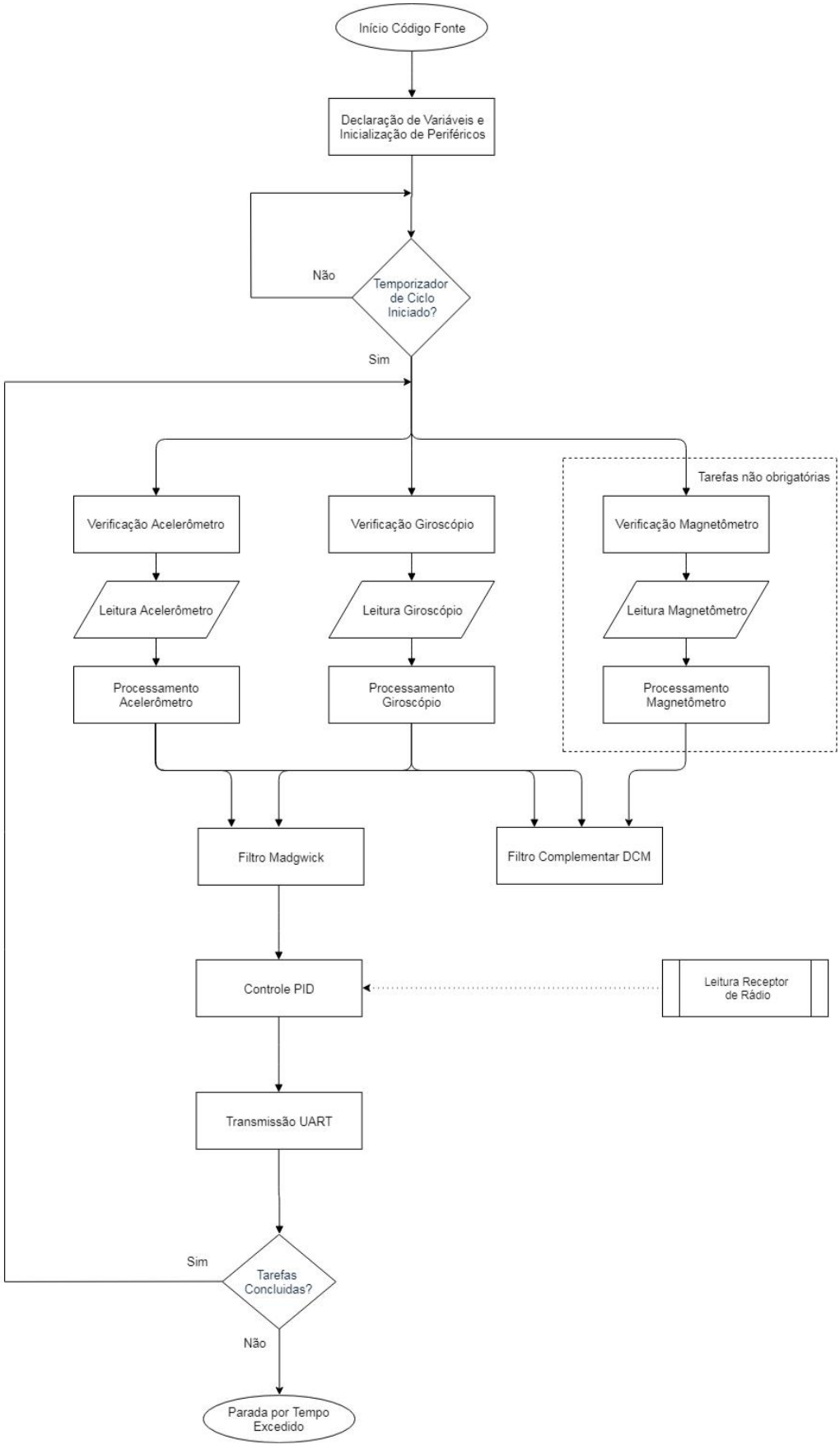
Colocar em operação um sistema com amostragem de dados sem o devido monitoramento de tempo seria imprudente, poderia condenar a coerência dos cálculos. Assim, optou-se na criação de um temporizador interno com a função de determinar o início de um ciclo de processamento e término do ciclo anterior. Para garantir que o ciclo seja completo, a função de interrupção do temporizador de ciclo verifica a conclusão da última tarefa. Questão suficiente na validação da execução completa, já que, as tarefas são intertravadas pelo algoritmo

de monitoramento de tarefas. Esta rotina ocorre em 390 Hz, frequência sutilmente inferior a amostragem dos sensores digitais, 400 Hz.

No temporizador de ciclo são habilitadas as primeiras tarefas. Elas verificam os sensores e, quando atendidas as condições, permitem as tarefas seguintes. Após a execução das tarefas pertinentes aos dados de sensoriamento (abordados na seção 4.3.2), agrupam-se as informações para realizar o condicionamento nos filtros (abordados na seção 4.4). Os resultados obtidos são então passados para a etapa que determina os valores de controle da posição dos eixos (abordados na seção 4.5).

No final, os dados do processo que ficam armazenados na memória podem ser passados ao terminal por meio do protocolo UART (*Universal Asynchronous Receiver Transmitter*). Neste projeto é empregada a biblioteca de funções UARTstdio [40] que compõe o pacote de utilitários para desenvolvimento de sistemas embarcados.

Figura 19 - Diagrama de fluxo das tarefas



Fonte: Autoria Própria

4.3 AQUISIÇÃO DE DADOS

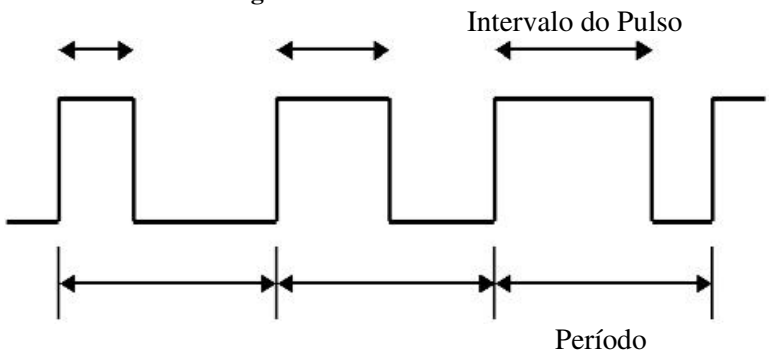
Após a concepção da plataforma de estudo, inicia-se a coleta dos dados, imprescindível no processo de estudo e compreensão preliminar do sistema.

4.3.1 Leitura do Receptor de Rádio

O receptor de rádio frequência é responsável por detectar as ondas de rádio enviadas e convertê-las em oitos canais de saída digital PWM [41].

O sinal de onda retangular PWM caracteriza-se por transferir informações através do intervalo de tempo que permanece no nível alto, ou seja, a diferença de tempo entre a borda de subida e a borda de descida [42]. A forma característica do sinal pode ser observada na Figura 20.

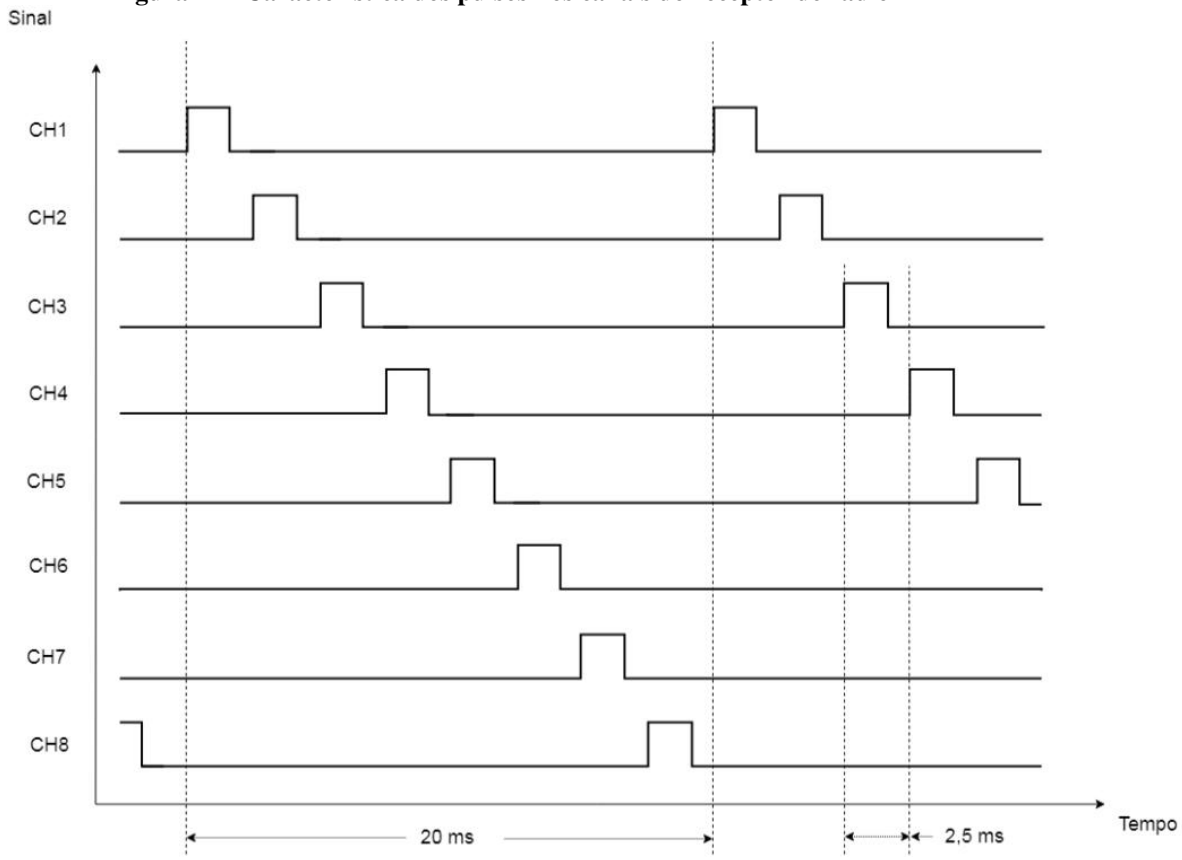
Figura 20 - Sinal de onda retangular PWM



Fonte: Adaptado de [43]

O ciclo de atualização ocorre a cada 20 milissegundos, correspondendo a 50 Hz de frequência. O pulso de cada canal acontece singularmente no período completo, ou seja, o nível alto existe alternadamente em um intervalo de 2,5 milissegundos, como pode ser visto na Figura 21.

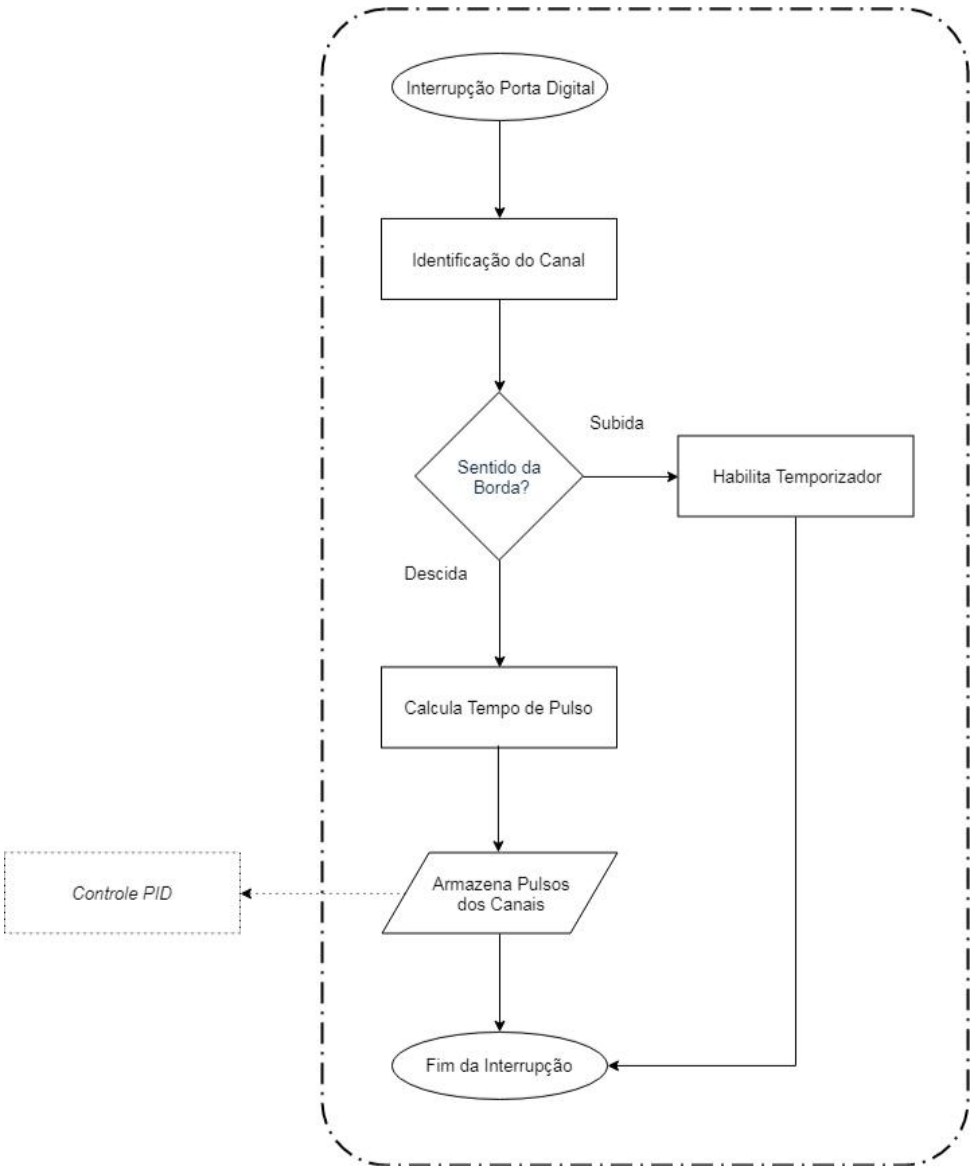
Figura 21 - Característica dos pulsos nos canais do receptor de rádio



Fonte: Autoria Própria

A leitura dos canais do rádio não entra na rotina principal de execução de tarefas vista na seção 4.2, pois a geração dos sinais é realizada pelo receptor de rádio, deste modo não é possível que o microcontrolador sincronize com os eventos do ciclo principal. Outra diferença
é sua orientação baseada em interrupções, assim o sistema deve capturar as bordas de subida e descida no exato instante em que ocorrem. A representação das etapas deste processo pode ser vista na Figura 22.

Figura 22 - Diagrama de fluxo da leitura do receptor de rádio

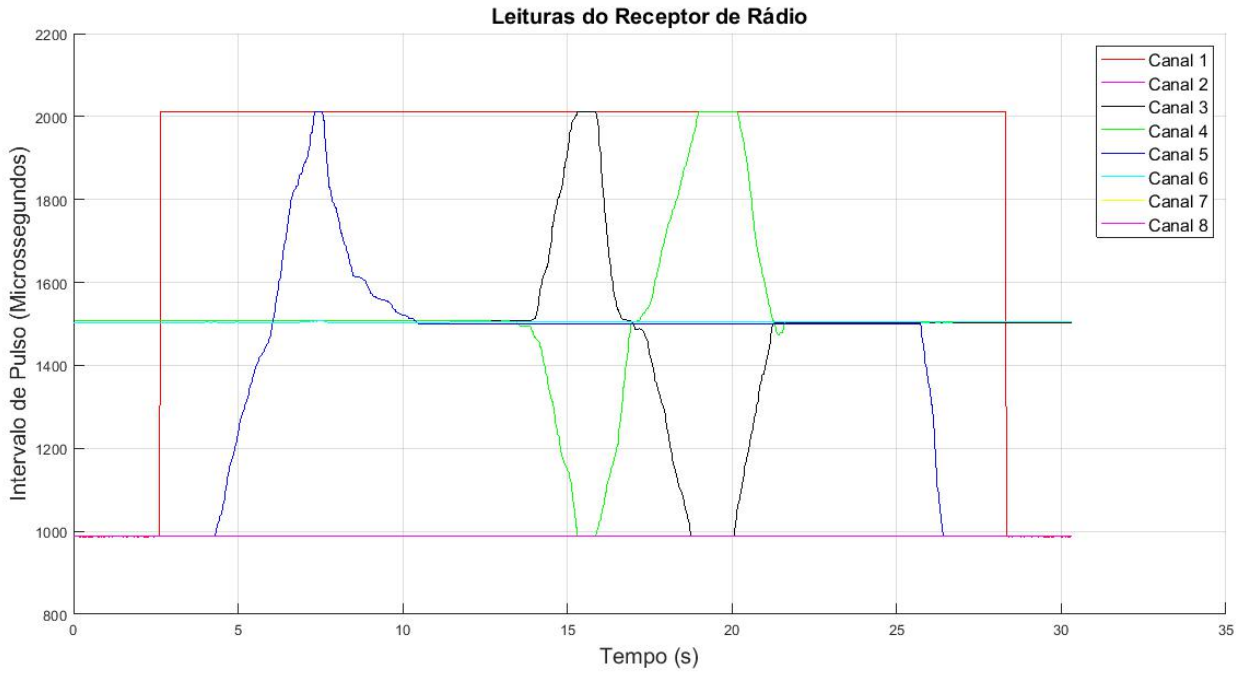


Fonte: Autoria própria

Os sinais capturados no receptor de rádio têm pulsos de tempo com valor mínimo de 988 microssegundos e valor máximo de 2012 microssegundos. Quando em posição neutra os canais têm 1500 microssegundos de pulso.

Algumas operações simples são realizadas no ensaio representado no gráfico da Figura 23. A primeira etapa é alterar o canal 1 do valor mínimo para o valor máximo, este comando aciona os motores. No segundo momento o canal 5 é ajustado para regular a porcentagem de aceleração. Na sequência são alterados simultaneamente os valores de referência nos eixos X e Y através dos canais 3 e 4, respectivamente. Por fim, a aceleração é reduzida para o mínimo e os motores desabilitados para encerar o teste. Os canais 2, 7 e 8 permanecem no valor mínimo e sem alteração durante o ensaio. O canal 6 permanece no valor médio sem alteração.

Figura 23 - Leituras do receptor de rádio



Fonte: Autoria Própria

Portanto, os valores lidos nos canais do receptor de rádio são utilizados pelo sistema para determinar a posição angular de referência (4 canais), ativar os motores na rotação mínima (1 canal) e ajustar os valores de ganho do controlador PID (3 canais). O Quadro 2 sintetiza qual a utilização dos comandos enviados remotamente ao quadrotor.

Quadro 2 - Utilização dos canais do receptor de rádio

Canal do Receptor	Porta do Microcontrolador	Função de Operação
CH1	PE2	Acionar motores
CH2	PE3	Ganho derivativo (K_d)
CH3	PA2	Referência do eixo X
CH4	PA3	Referência do eixo Y
CH5	PA4	Porcentagem de aceleração
CH6	PA5	Referência do eixo Z
CH7	PA6	Ganho proporcional (K_p)
CH8	PA7	Ganho integral (K_i)

Fonte: Autoria própria

4.3.2 Leitura dos Sensores Inerciais

Para adquirir conhecimento sobre as particularidades de cada sensor foram realizados testes individuais para avaliar a necessidade de filtragem dos dados coletados. As principais características serão apresentadas e as configurações pertinentes serão discutidas a fim de aumentar a confiabilidade do sistema.

4.3.2.1 Caracterização do Acelerômetro

O acelerômetro ADXL345 possui um microcontrolador integrado, responsável pelo gerenciamento do sistema de amostragem e pelo armazenamento dos dados recolhidos. As medidas de cada eixo ficam armazenadas em dois registradores de oito bits no formato computacional complementar de dois. Um registrador de estado é verificado com a finalidade de conhecer o momento que novos dados são obtidos, para então realizar a leitura ininterrupta dos seis registradores de dados.

As informações contidas no acelerômetro são transmitidas através do protocolo de comunicação digital I^2C, sob o endereço de memória 53_{hex}, operando com 3,3 Volts nos barramentos. Há no *datasheet* do dispositivo [29], informações detalhadas sobre o funcionamento da comunicação I^2C, além de explicações sobre as diversas funcionalidades referentes a interrupção, modo de operação e taxa de amostragem.

Na inicialização do sistema o acelerômetro deve ser configurado para operar conforme os objetivos pretendidos. Para isto, comandos específicos podem ser enviados ao dispositivo para gravar as configurações desejadas em registradores de 8 bits integrados. O Quadro 3 relaciona os registradores de configuração, seu endereço no mapa de memória do dispositivo, o valor que foi gravado e a configuração realizada.

Quadro 3 - Configurações de inicialização do acelerômetro

Nome do Registrador	Endereço	Valor Escrito	Descrição
BW_RATE	$2C_{hex}$	$0C_{hex}$	Configuração frequência de amostragem
POWER_CTL	$2D_{hex}$	08_{hex}	Seleção modo de operação
INT_ENABLE	$2E_{hex}$	00_{hex}	Ajuste de opções de interrupção
DATA_FORMAT	31_{hex}	$0B_{hex}$	Configuração do formato dos dados

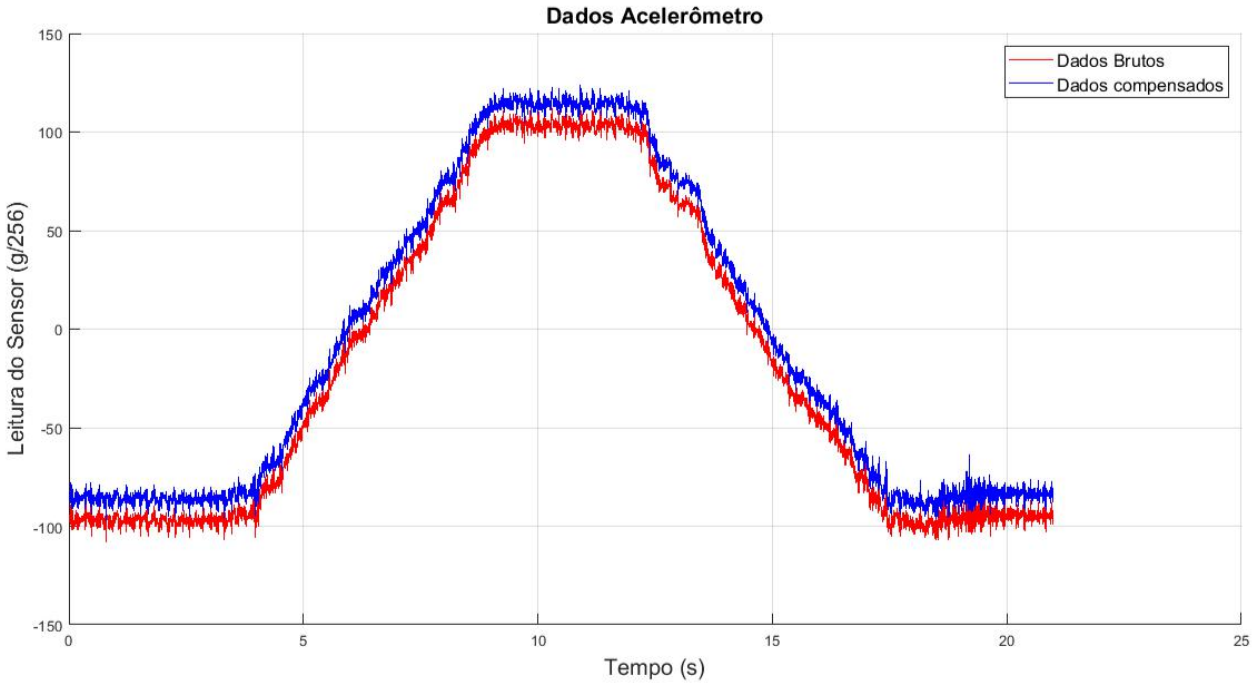
Fonte: Autoria própria

No registrador BW_RATE ($2C_{hex}$) o valor gravado ($0C_{hex}$) faz com que a frequência de amostragem seja 400 Hz. A configuração efetuada no registrador POWER_CTL ($2D_{hex}$) faz o dispositivo obter medidas continuamente, na taxa de amostragem definida em BW_RATE e sem a possibilidade de entrar em modo de espera. A função do registrador INT_ENABLE ($2E_{hex}$) é selecionar os recursos de interrupção que podem ser gerados, o valor passado (00_{hex}) não ativa nenhuma função de interrupção extra, apenas ocorrem interrupções quando novos dados medidos estão prontos para a leitura. O registrador DATA_FORMAT (31_{hex}) elege através do valor transmitido a resolução de 13 bits, justificado a direita com relação aos dois registradores de dados e com fundo de escala em duas unidades de gravidade (±2 g).

Neste trabalho é adotado um método empírico com alguns movimentos simples, diferente dos movimentos robóticos e programados utilizados em Carvalho et al. [44]. Ao longo do texto o método será denominado Nicolaus, na qual obedece ao seguinte procedimento: inicia-se o teste com o eixo imóvel por aproximadamente 3 segundos na posição angular aproximada de 20°; na sequência é rotacionado por aproximadamente 6 segundos para o lado contrário; onde é mantido imóvel na posição aproximada de -30° por aproximadamente 3 segundos; então é realizado o movimento de retorno para a posição inicial por aproximadamente 6 segundos; e por fim, alocado na posição original aproximada de 20° por aproximadamente três segundos.

O resultado do teste realizado no acelerômetro pode ser visto no gráfico da Figura 24, onde é apresentado o dado de saída bruto e compensado (calibrado), ambos em função do tempo para um dos eixos. A compensação ou calibração é obtida com o objetivo de eliminar um desvio sistemático proveniente do processo de fabricação. Detalhes do processo de calibração podem ser encontrados no manual do dispositivo [29].

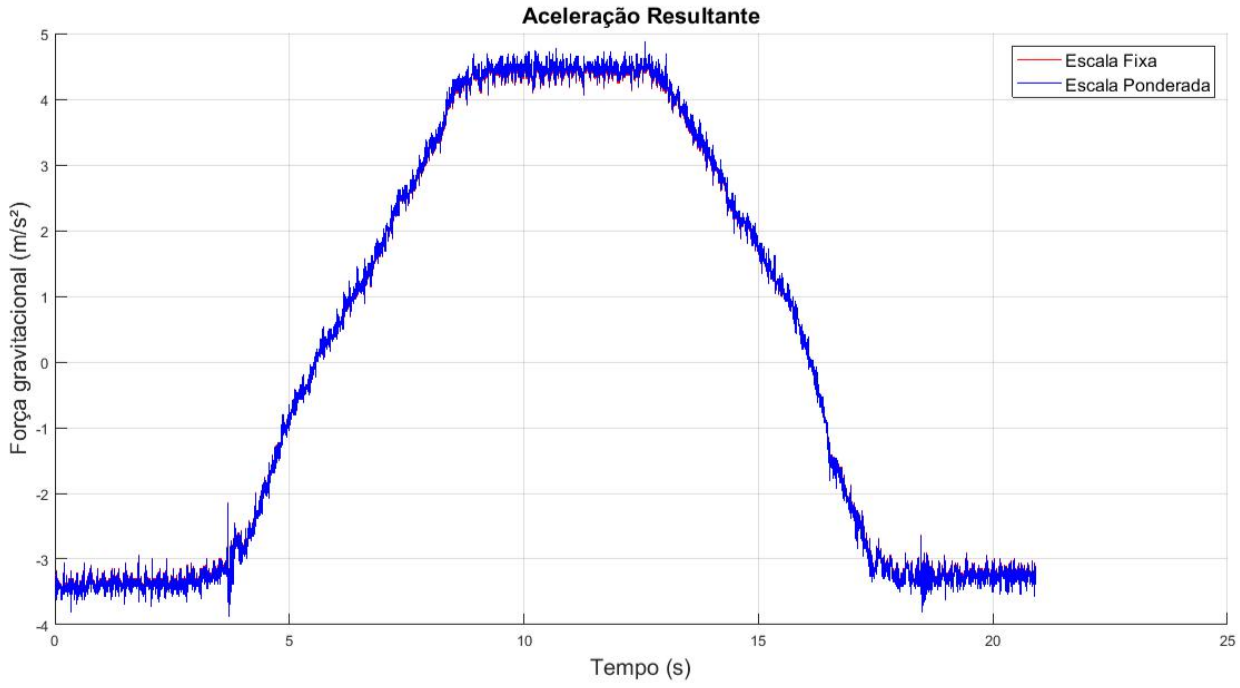
Figura 24 - Dados acelerômetro



Fonte: Autoria Própria

Para encontrar efetivamente a aceleração os dados brutos lidos no sensor devem ser multiplicados por um fator escalar. Esse fator depende da resolução digital configurada e pode ser encontrado no *datasheet* do dispositivo [31]. Outra maneira é utilizar um recurso extra da calibração que produz valores específicos para cada eixo, basta dividir o valor da aceleração gravitacional (9,81 m/s) pelo valor máximo obtido nas leituras do acelerômetro. No eixo estudado ambos resultam em valores numericamente próximos e há sobreposição dos dados, como pode ser visto na Figura 25, indicando assim medidas autênticas.

Figura 25 - Aceleração resultante



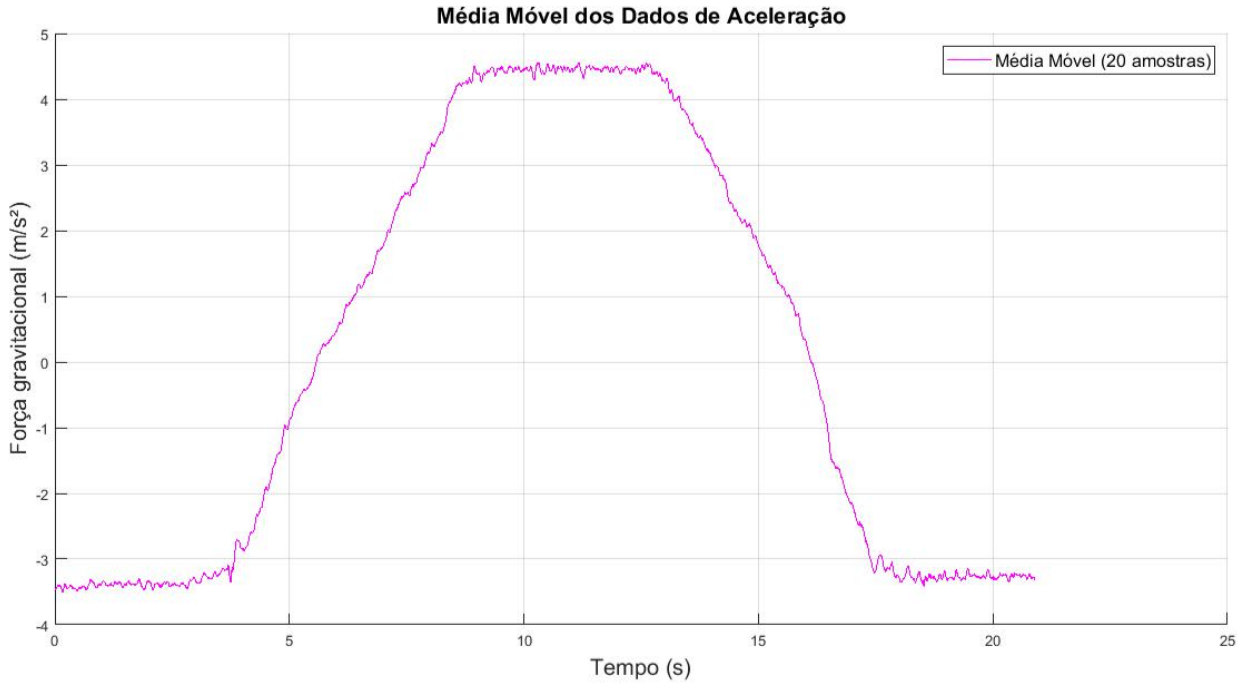
Fonte: Autoria Própria

4.3.2.1.1 Média Móvel

É recomendado, no *datasheet* calcular a média entre os dados obtidos do acelerômetro durante um intervalo de tempo igual a 0,1 segundos, para as taxas de amostragem maiores que 10 Hz [29]. Como o sensor opera em 400 Hz deveriam ser utilizadas 40 amostras, no entanto o microcontrolador suportou armazenar apenas 20 amostras, mas que foram suficientes para reduzir o nível de ruído do sinal, assim como pode ser observado na Figura 26. O cálculo é realizado segundo a equação (1).

$$m = \frac{\sum_{i=0}^{20} a[n-i]}{20} \quad (1)$$

Figura 26 - Média móvel dos dados de aceleração



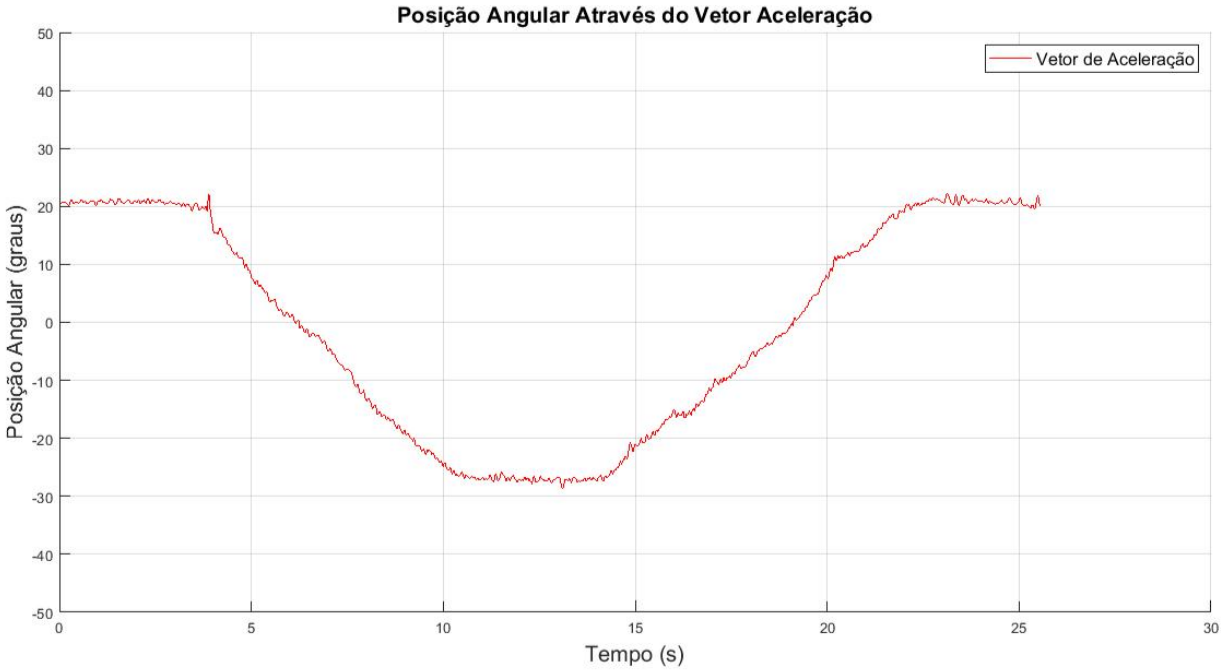
Fonte: Autoria Própria

4.3.2.1.2 Vetor Aceleração

Para calcular o ângulo de inclinação de um eixo através dos valores do vetor de aceleração gravitacional utiliza-se o método apresentado em [45]. Primeiramente efetua-se o cálculo do módulo do vetor de aceleração. Em seguida, para obter o valor angular do eixo, calcula-se o arco cosseno entre a componente do eixo de interesse e o módulo. Para seguir o padrão dos filtros estudados na seção 4.4 o valor calculado troca de sinal, conforme a equação (2). Com os motores desligados tem-se o resultado da Figura 27. Quando os motores estão ligados o nível de ruído eleva-se em função da vibração produzida pelo conjunto de atuadores, como pode ser visto na Figura 28.

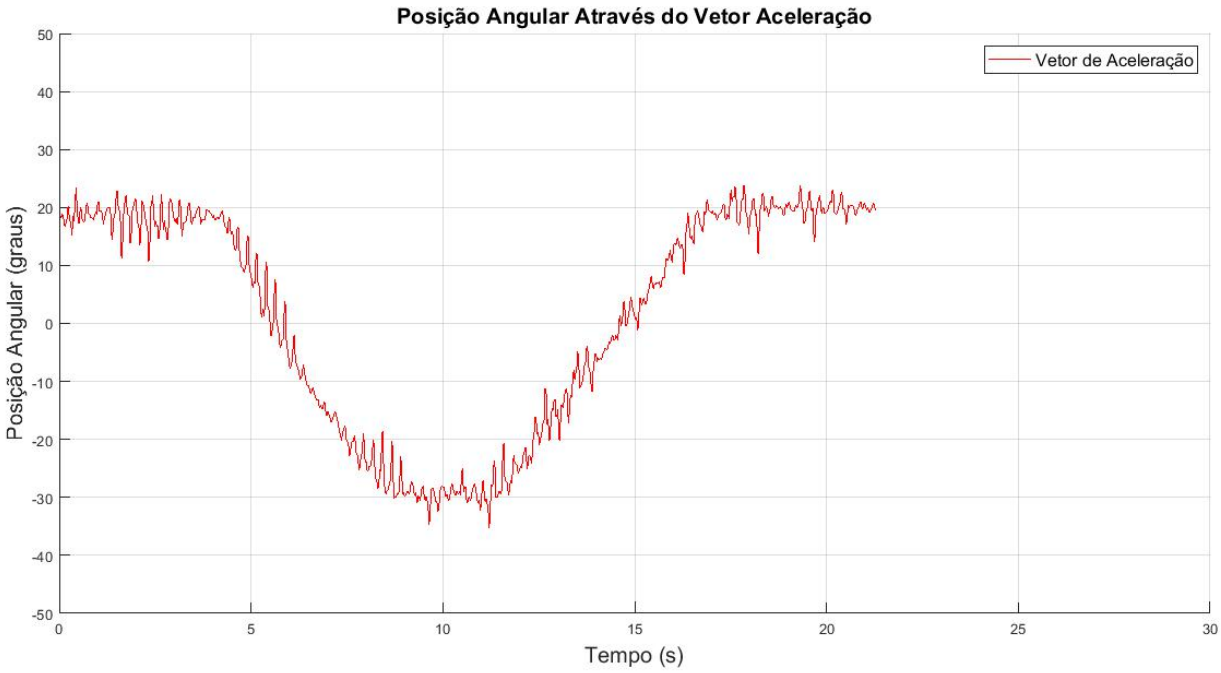
$$\theta = -\tan^{-1}\frac{y[n]}{\sqrt{x[n]^2 + y[n]^2}} \quad (2)$$

Figura 27 - Posição angular com motores desligados



Fonte: Autoria Própria

Figura 28 - Posição angular com motores ligados



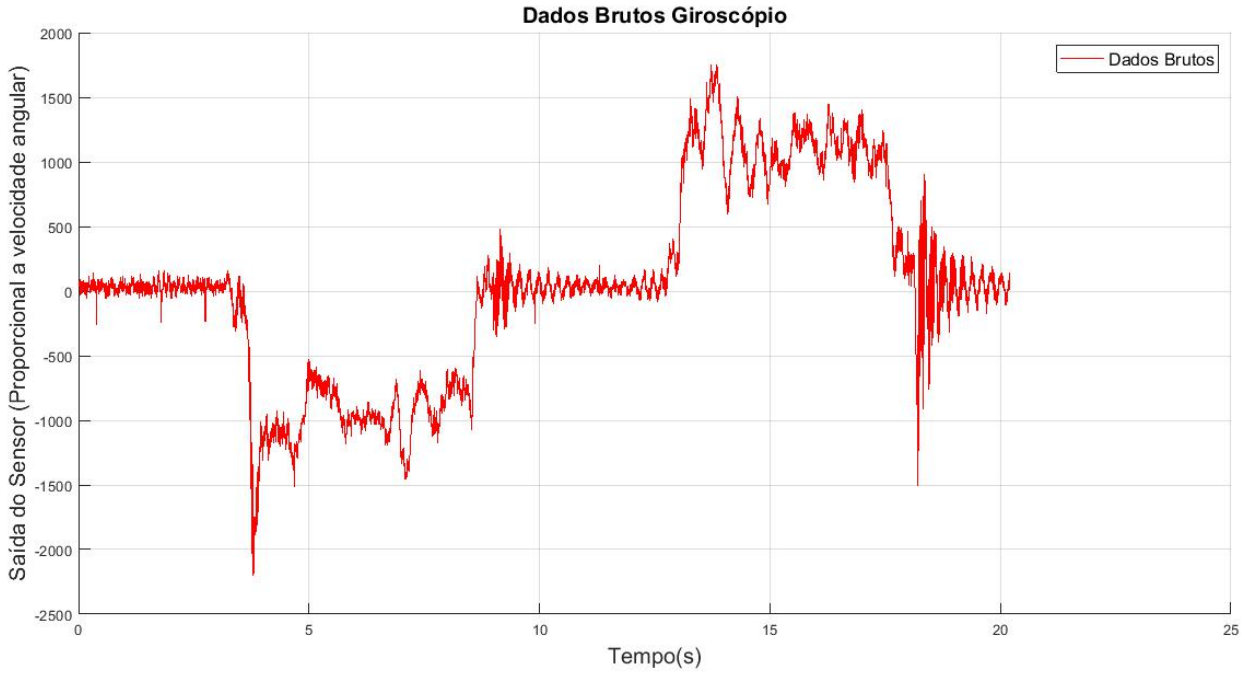
Fonte: Autoria Própria

4.3.2.2 Caracterização do Giroscópio

Os dados coletados pelo giroscópio são disponibilizados para leitura no mesmo formato do acelerômetro, dois registradores para cada eixo armazenados no formato complementar de dois com escala completa. A leitura dos dados é realizada sob o protocolo I^2C, no mesmo barramento do acelerômetro com 3,3 V. O endereço escravo deste sensor digital é 69_{hex}, informação obtida no *datasheet* do dispositivo [32]. Esse *datasheet* traz detalhes específicos do funcionamento da comunicação I^2C, esclarece sobre a utilização de modos de operação diferenciados e recursos para controlar a geração de interrupções.

Para avaliar a medição realizada por este sensor, o mesmo procedimento de teste aplicado no acelerômetro é efetuado, seguindo assim o método Nicolaus. Embora sejam os mesmos movimentos o modo de interpretar os resultados é diferente, visto que o giroscópio realiza a medição da velocidade angular. Enquanto o eixo analisado estiver parado, independentemente da posição, o sensor deve medir valores próximos de zero. Nos intervalos de tempo em que existe rotação, o giroscópio quantifica esse movimento realizando a leitura da velocidade angular individualmente no eixo de interesse. Na Figura 29 são exibidos os dados brutos coletados diretamente do sensor durante a realização do procedimento de teste padrão.

Figura 29 - Dados brutos giroscópio

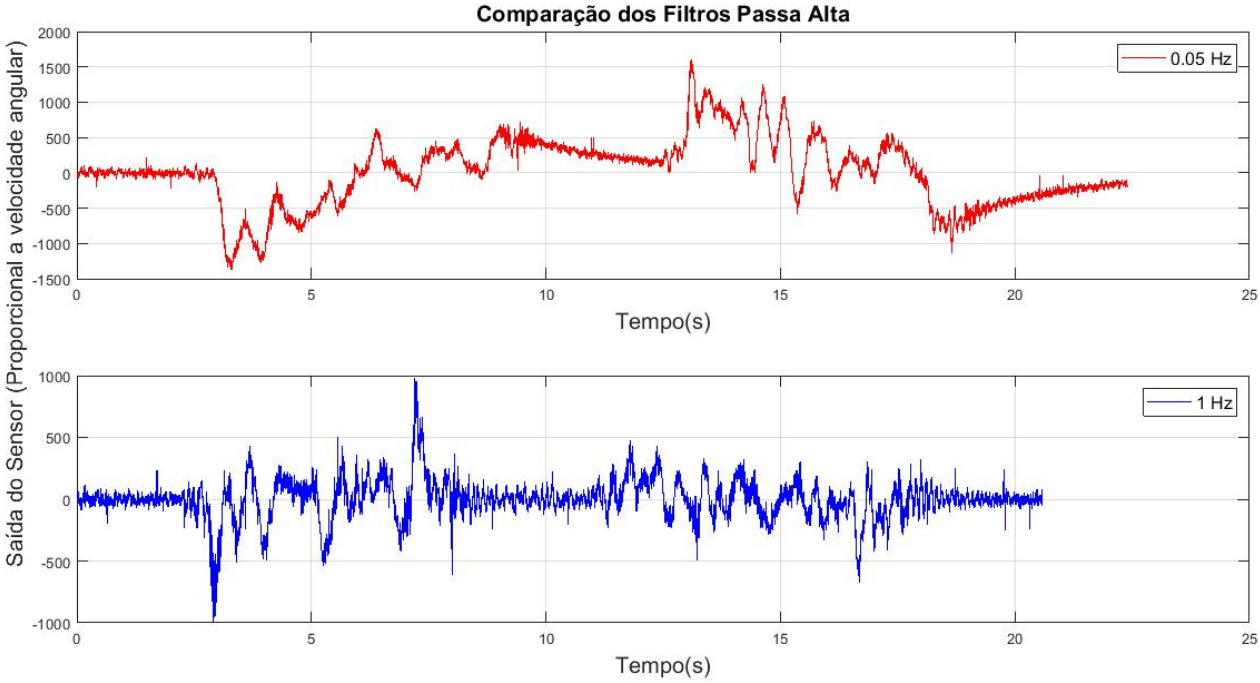


Fonte: Autoria Própria

Na busca de um método para retirar a componente contínua no tempo, também chamada nos giroscópios de tendência à deriva, aplica-se o filtro passa-alta integrado ao giroscópio [32]. Não existem informações disponibilizadas pelo fabricante sobre características técnicas dos filtros, tal como a tipologia e a ordem do filtro.

Foram realizados dois testes selecionando primeiro a frequência de corte em 0,05 Hz e depois em 1 Hz, conforme os gráficos da Figura 30. Com a aplicação do filtro passa-alta de 0,05 Hz os sinais foram atenuados quando o movimento estava constante e o tempo de convergência mostrou-se muito elevado para essa aplicação. Já o filtro passa alta de 1 Hz provocou distorções incompreensíveis, detectando apenas movimentos abruptos e sem significado. A interpretação dos resultados esclarece que o filtro passa-alta integrado ao sensor não é adequada ao protótipo desenvolvido neste trabalho.

Figura 30 - Filtros passa-alta do giroscópio

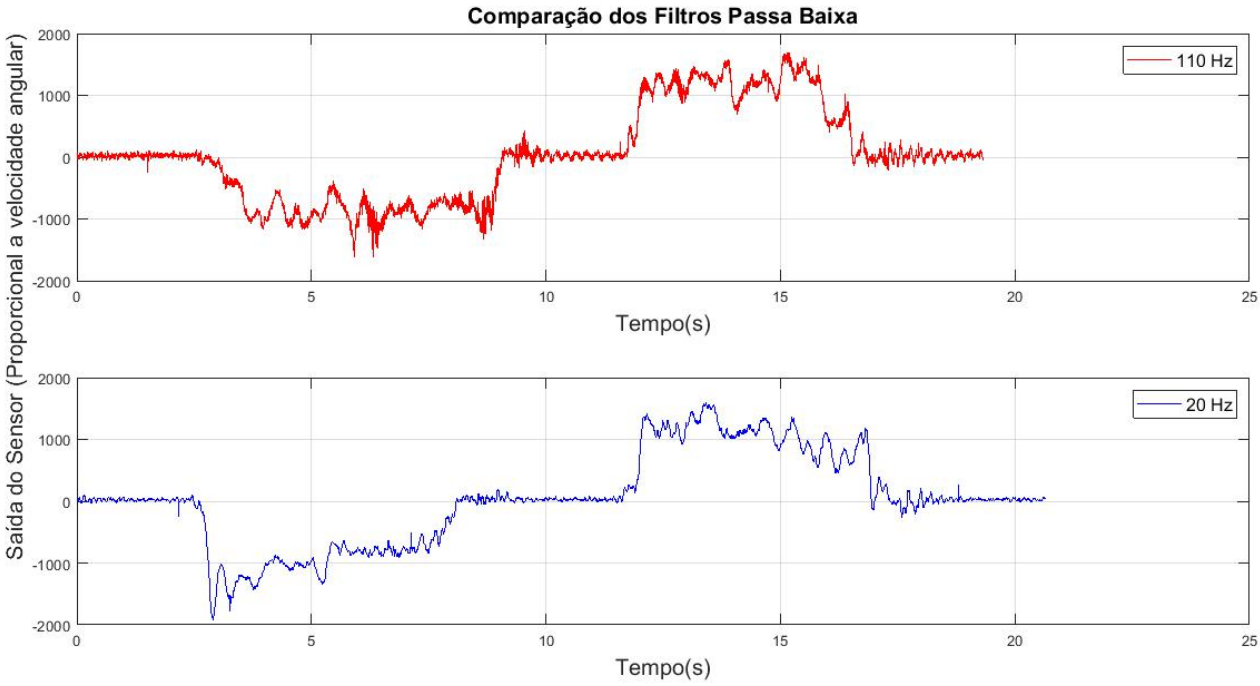


Fonte: Autoria Própria

O excesso de ruído é extremamente prejudicial ao desempenho do sistema. Na tentativa de reduzir sua influência aplicou-se o filtro passa-baixa integrado ao sensor nas frequências de corte 110 Hz e 20 Hz, como pode ser visto na Figura 31. No primeiro, com 110 Hz, o sinal é semelhante aos dados brutos do sensor. Em contrapartida o segundo, com 20 Hz, o sinal foi atenuado significativamente sem prejudicar o intervalo tempo na qual o valor de

saída converge. Estas características são valiosas para o sistema, portanto o filtro passa-baixa de 20 Hz é utilizado nas próximas etapas de desenvolvimento deste projeto.

Figura 31 - Filtros passa-baixa do giroscópio



Fonte: Autoria Própria

Com a caracterização do giroscópio, é possível determinar a configuração de inicialização do dispositivo. As configurações são realizadas através da gravação de valores específicos em registradores integrados. A relação dos registradores e das configurações realizadas podem ser vistas no Quadro 4.

Quadro 4 - Configurações de inicialização do giroscópio

Nome do Registrador	Endereço	Valor Escrito	Descrição
CTLR_REG1	20_{hex}	$8F_{hex}$	Configuração da taxa de amostragem
CTLR_REG2	21_{hex}	00_{hex}	Configurações do filtro passa-alta
CTLR_REG4	23_{hex}	80_{hex}	Seleção do fundo de escala
CTLR_REG5	24_{hex}	02_{hex}	Configuração e habilitação dos filtros
FIFO_CTLR_REG	$2E_{hex}$	00_{hex}	Configuração do modo da fila de dados

Fonte: Autoria própria

O registrador CTLR_REG1 (20_{hex}) configura através do valor gravado ($8F_{hex}$) a taxa de amostragem para 400 Hz, seleciona a frequência de corte do filtro passa-baixa para 20 Hz e habilita a medição em todos os eixos no modo normal de consumo de energia. No registrador CTLR_REG2 (21_{hex}) estão disponíveis as configurações do filtro passa-alta, esse filtro não será utilizado, portanto o valor escrito é irrelevante. No registrador CTLR_REG4 (23_{hex}) o valor gravado (80_{hex}) configura o fundo de escala em 250 g/s. No registrador CTLR_REG5 (24_{hex}) é escrito o valor (02_{hex}) responsável pela ativação do filtro passa-baixa, esse filtro teve a frequência de corte configurada no registrado CTLR_REG1. No registrador FIFO_CTLR_REG ($2E_{hex}$) é transmitida uma configuração (00_{hex}) para não utilizar a fila de dados.

4.3.2.3 Caracterização do Magnetômetro

O magnetômetro HMC5883L realiza a comunicação com a placa microcontrolada através da interface digital I^2C, sob o endereço $1E_{hex}$, conectado no mesmo barramento de 3,3 V do acelerômetro e giroscópio utilizados neste projeto. Os dados de cada eixo estão disponíveis em dois registradores que possuem juntos 16 bits no formato complementar de dois. O *datasheet* do sensor [32] especifica os endereços dos dados de cada eixo, além de elucidar questões referentes a interrupções, taxas de amostragem e recursos de operação.

No início do sistema as configurações do Quadro 5 são gravadas em registradores específicos para realizar a configuração do sensor.

Quadro 5 - Configurações de inicialização do magnetômetro

Nome do Registrador	Endereço	Valor Escrito	Descrição
Configuration Register A	00_{hex}	58_{hex}	Configuração da taxa de amostragem
Configuration Register B	01_{hex}	20_{hex}	Configuração do fundo de escala
Mode Register	02_{hex}	00_{hex}	Seleção do modo de operação

Fonte: Autoria própria

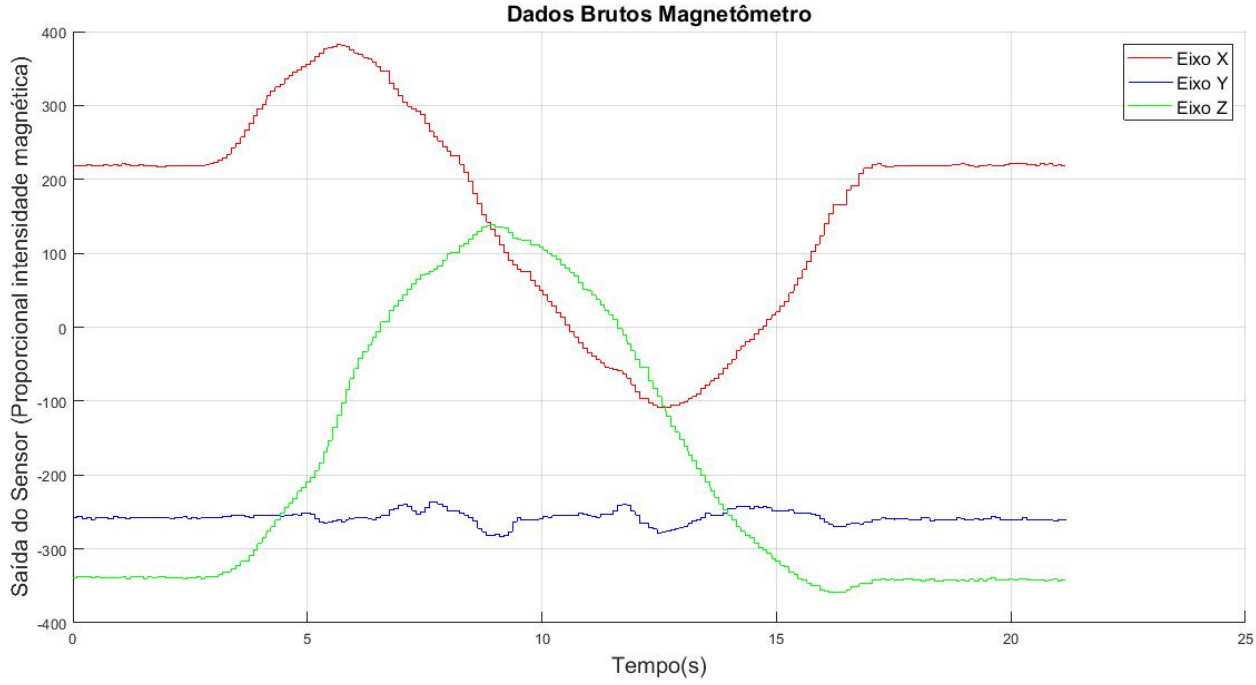
O valor escrito (58_{hex}) no registrador Configuration Register A (00_{hex}) é responsável por configurar a taxa de amostragem para o maior valor disponível, 75 Hz, determinar o modo normal de medição e selecionar a quantidade de 4 amostras para calcular a média aritmética que é disponibilizada nos registradores de saída do sensor. O registrador Configuration Register B (01_{hex}) com o valor escrito (20_{hex}) elege o ganho do dispositivo para 0,88 Gauss. O registrador

Mode Register (02_{hex}) determina, com o valor gravado (00_{hex}), o modo de operação contínuo para que sejam realizadas medidas constantemente na frequência configurada no Configuracion Register A.

Para conhecer as medidas de fluxo magnético que o sensor obtém em cada eixo, foram plotadas as informações dos três eixos simultaneamente, como pode ser visto na Figura 23. O procedimento experimental inicia com o modelo imóvel por aproximadamente 3 segundos com a frente apontada para o norte geográfico; então é rotacionado no ângulo de guinada no sentido horário 360° por aproximadamente 12 segundos retornando à posição inicial; e por fim é mantido imóvel por aproximadamente 3 segundos.

Analisando somente o eixo X da Figura 32, constata-se que a intensidade de fluxo magnético aumenta até ficar totalmente paralela ao campo magnético terrestre, e na sequência diminui para o valor inicial. Na sequência do estudo sobre o magnetômetro serão considerados os dados nos eixos X e Z, pois a variação dos valores mensurados é significativa.

Figura 32 - Dados brutos do magnetômetro

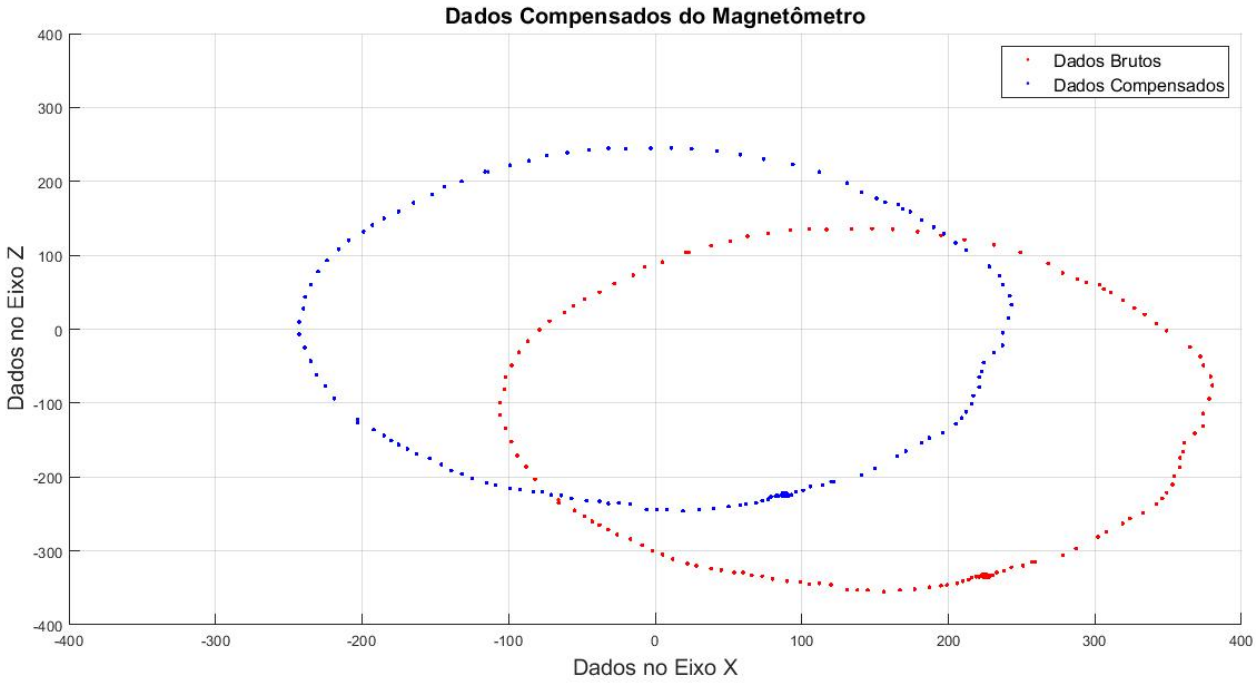


Fonte: Autoria Própria

No processo de fabricação do magnetômetro podem surgir desvios sistemáticos que diferem em cada eixo. Para corrigir executa-se um processo de calibração simples, na qual são identificados os valores máximos e mínimos de medição, e assim, calculado o valor médio que representa uma medição nula, os detalhes do processo de calibração podem ser encontrados em

[46] [47]. O resultado da calibração pode ser visto no gráfico da Figura 33, onde são apresentados os dados de saída brutos e compensados (calibrados).

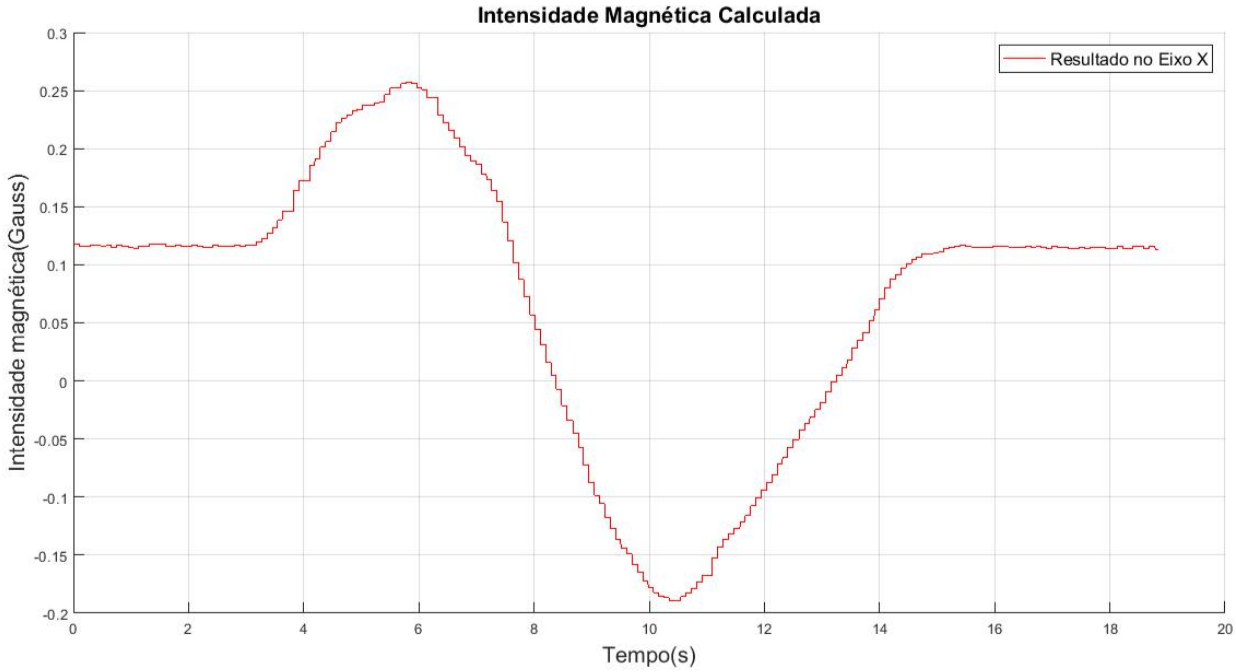
Figura 33 - Dados compensados do magnetômetro



Fonte: Autoria Própria

Nas configurações de inicialização o magnetômetro foi configurado com a resolução ± 1,3 Gauss. Conforme informações contidas no *datasheet* do dispositivo multiplicando-se os dados compensados pela resolução digital 0,92, obtém-se como resultado a intensidade magnética na unidade física Gauss, como pode ser visto no gráfico da Figura 34.

Figura 34 - Intensidade magnética calculada



Fonte: Autoria Própria

4.4 FUSÃO DE SENSORES

Ficou demonstrado no capítulo anterior que os sensores inerciais, principalmente os que mensuram aceleração e giro, contém intrinsicamente altos níveis de ruído. Em virtude disso, algum artificio computacional deve ser aplicado para condicionar os sinais [44]. Para obter a atitude do quadrotor através destes sensores utiliza-se estimadores em conjunto com fusão de sensores. Embora não seja objeto de estudo nesse trabalho, o Filtro de Kalman Estendido [48] (EKF, sigla inglesa para *Extended Kalman Filter*) foi utilizado em numerosos projetos com resultados satisfatórios [49] [50]. Para aplicações semelhantes ao deste projeto destacam-se o Filtro Complementar DCM (*Direction Cosine Matrix*) [51] e o Filtro Madgwick [27], estudados na sequência do trabalho. Embora chamados simplesmente de filtros, as implementações incluem sofisticados métodos de condicionamento de sinais e técnicas de estimação.

4.4.1 Filtro Complementar DCM

A matriz DCM (*Direction Cosine Matrix*) consiste no cosseno dos ângulos de todas as possíveis combinações entre a rotação relativa de um corpo e os versores globais. No trabalho de [51] são feitas diversas demonstrações teóricas que validam a utilização desta ferramenta na

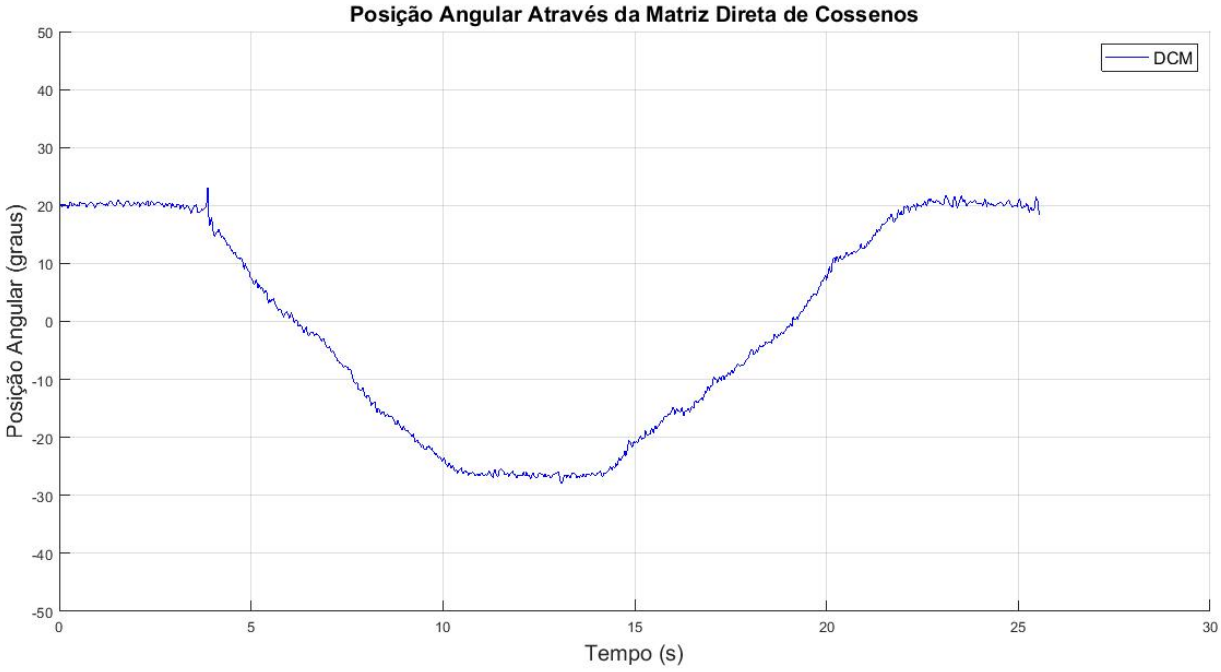
orientação cinemática. Em função da linearização presente na matriz DCM, sistemas de processamento em tempo real podem estimar a orientação de um corpo rígido com recursos computacionais reduzidos.

Já em [52] são apresentados os recursos aplicados para implementar a fusão dos dados entre o acelerômetro, giroscópio e magnetômetro, tal como a normalização de vetores e a divisão do erro entre os eixos para resultar em um valor intermediário.

Entre os recursos disponibilizados pela Texas Instruments para os microcontroladores da série TM4C, está a TivaWare^{TM} Sensor Library [38], uma coleção de funções e drivers para interação com sensores de ambiente. Dentre eles, será utilizado neste estudo o módulo Filtro Complementar DCM (CompDCM), projetado sob as considerações de [52].

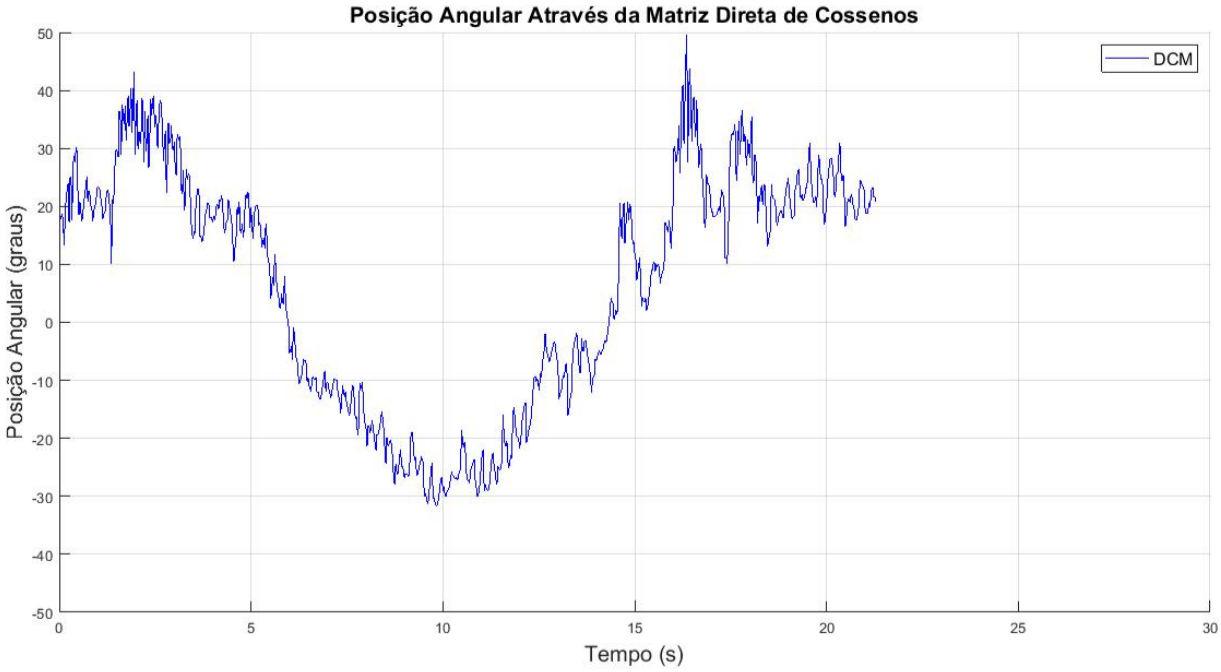
Seguindo o padrão experimental do método Nicolaus, realizam-se dois testes para avaliar o filtro DCM. Na Figura 35 é apresentado o resultado da estimação com os motores desligados e na Figura 36 o resultado com os motores do protótipo ligados. Em função da intensidade dos ruídos presentes, o filtro complementar DCM é considerado ineficaz para este projeto.

Figura 35 - Resposta filtro DCM com motores desligados



Fonte: Autoria Própria

Figura 36 - Resposta filtro DCM com motores ligados

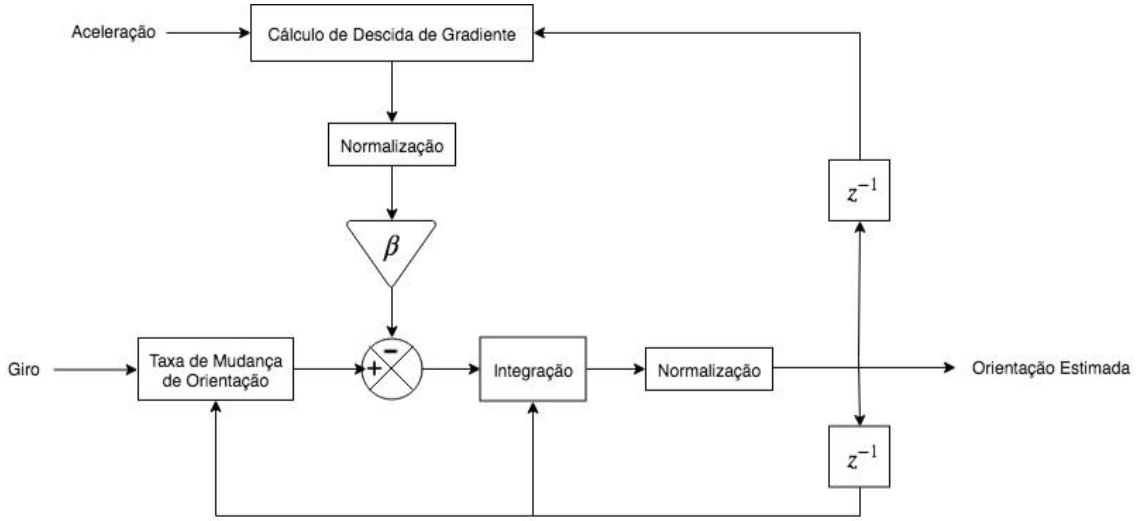


Fonte: Autoria Própria

4.4.2 Filtro Madgwick

A técnica de estimação de atitude introduzida por Mahony [53] representa a rotação de um corpo por quaterniões. A principal vantagem deste método é o custo computacional inexpressivo, pelas características dos quaterniões. Seguindo as mesmas premissas, Madgwick [27] realiza a fusão de velocidades angulares, acelerações, como pode ser visto na Figura 37. Inclui ainda, um novo conceito para calcular o erro utilizando um algoritmo derivado e otimizado da descida do gradiente.

Figura 37 - Diagrama de blocos do filtro Madgwick



Fonte: Adaptado de Michael Edvardsen e Joel Rietz [54]

O quaternião é um vetor de quatro elementos que pode ser convertido para qualquer sistema de coordenada de três dimensões [55]. O vetor é composto por um elemento numericamente real e três elementos numericamente complexos, com a capacidade de representar principalmente rotações e velocidades angulares.

No trabalho [27], Madgwick ajusta o ganho β do filtro no valor 0,041 para obter a melhor precisão, validada com ferramentas de medição ótica. A documentação e o código em linguagem C podem ser encontrados em [56]. A função do ganho β é adequar a componente da descida de gradiente advindo do acelerômetro à magnitude da mudança de orientação do giroscópio. Em outras aplicações, como em multirotores, Madgwick recomenda a realização de experiências para alcançar o melhor ajuste.

Neste estudo em particular, as medidas devem combinar exatidão e rapidez na convergência do sinal, portanto o ganho β foi avaliado em tempo real na faixa entre 0,01 e 2,00. Foi constatado visualmente que em 0,1 a resposta do filtro fica ligeiramente atrasada em relação ao movimento real. Já com o valor do ganho β ajustado em 0,8 a resposta do filtro passa a apresentar oscilações relativamente elevadas. Deste modo, o desempenho adequado nas experiências práticas ocorreu com um valor intermediário para o ganho β, igual a 0,4.

São apresentados os resultados dos ângulos de arfagem ( ) e rolamento ( ), pela semelhança analítica. O ângulo de guinada ( ) não é avaliado neste trabalho por quatro fatores: os testes são realizados em ambientes fechados que podem conter distorções na direção do campo magnético; há peças metálicas próximas ao sensor que podem provocar interferências;

o campo magnético na localidade dos testes tem baixa intensidade comparado a outros locais do globo terrestre; e, a posição do ângulo de guinada na obtenção da estabilidade é facultativo.

Quando comparado aos quaterniões, os ângulos de Euler são mais intuitivos e de matemática mais simples para representar a rotação de um corpo rígido e realizar seu controle. Assim sendo, realiza-se a conversão por meio das equações 3, 4 e 5, apresentadas em [57].

$$= - \quad (2\ 1\ 3 - 2\ 0\ 2) \tag{3}$$

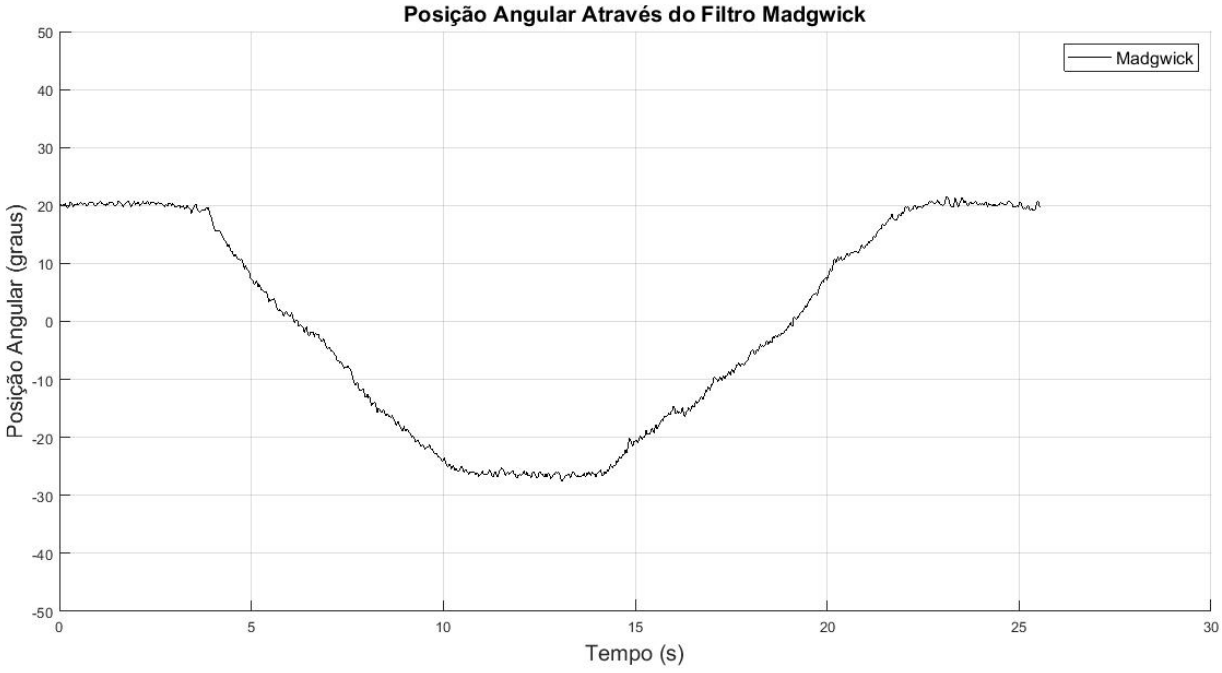
$$= \quad 2\,(2 \qquad\qquad + 2 \qquad , \qquad {}^{2}{}_{-} \qquad {}^{2}{}_{-} \qquad {}^{2}{}_{+} \qquad) \tag{4}$$

$$= \arctan 2\,(2 \qquad\qquad + 2 \qquad , \qquad {}^{2}{}_{+} \qquad {}^{2}{}_{-} \qquad {}^{2}{}_{-} \qquad) \tag{5}$$

Na Figura 38 tem-se a posição angular obtida com o processamento do filtro Madgwick quando os motores do protótipo estão desligados e na Figura 39 quando os motores estão ligados. O procedimento de teste realizado é o método Nicolaus, idêntico ao aplicado para os sensores acelerômetro e giroscópio, como também para o filtro complementar DCM.

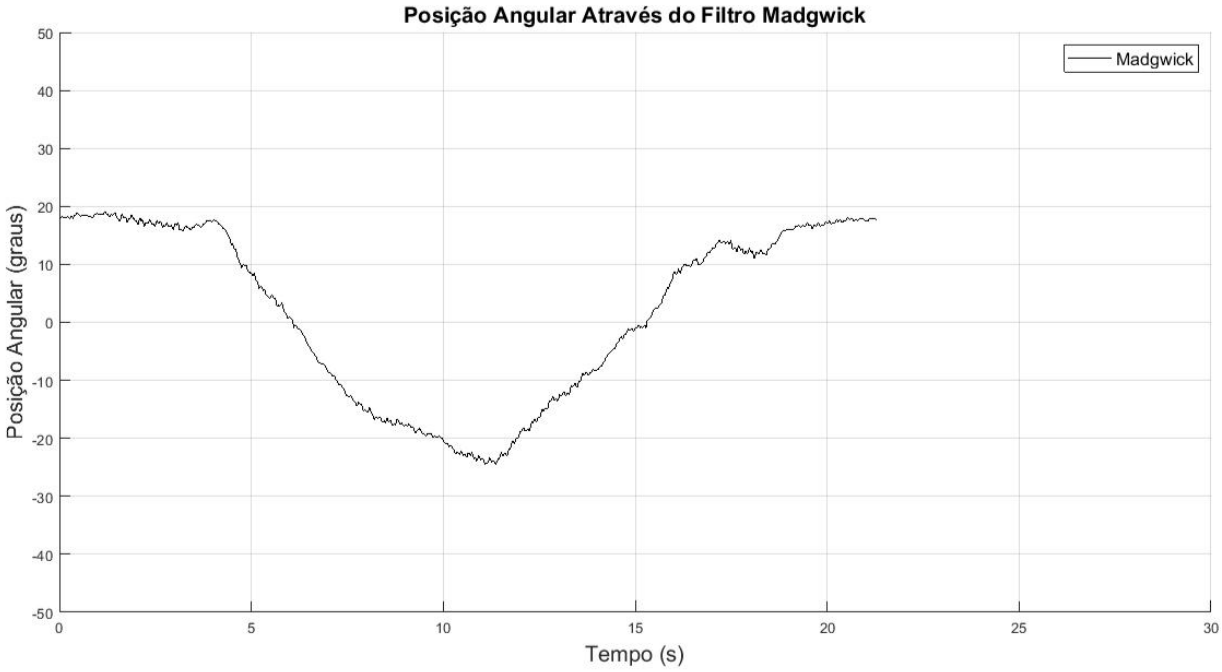
A resposta da fusão de filtros do método Madgwick alcançou notório desempenho na rejeição dos ruídos. Em contrapartida houve distorções suaves no valor estimado, sem prejudicar a funcionalidade do sistema, conforme será comprovado no próximo capítulo.

Figura 38 - Resposta filtro Madgwick com motores desligados



Fonte: Autoria Própria

Figura 39 - Resposta filtro Madgwick com motores ligados



Fonte: Autoria Própria

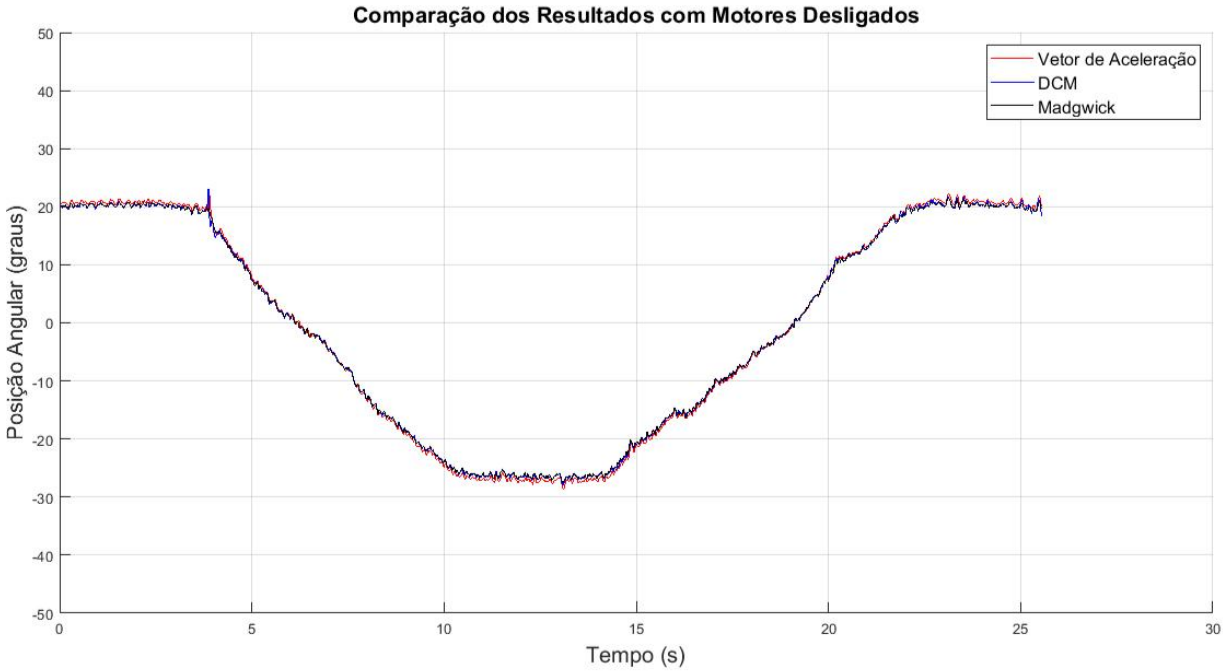
4.4.3 Comparação dos Métodos

Para comparar as técnicas de estimação angular aplicadas para um eixo do quadricóptero realizou-se um teste aplicando todos os métodos simultaneamente: medição com o vetor de aceleração, filtro complementar DCM e filtro Madgwick.

O teste seguiu o mesmo procedimento dos experimentos anteriores, chamado de método Nicolaus. Em função da comunicação serial ser limitada, a transmissão dos dados deste experimento é reduzida em um décimo, ou seja, são enviadas 39 amostras por segundo.

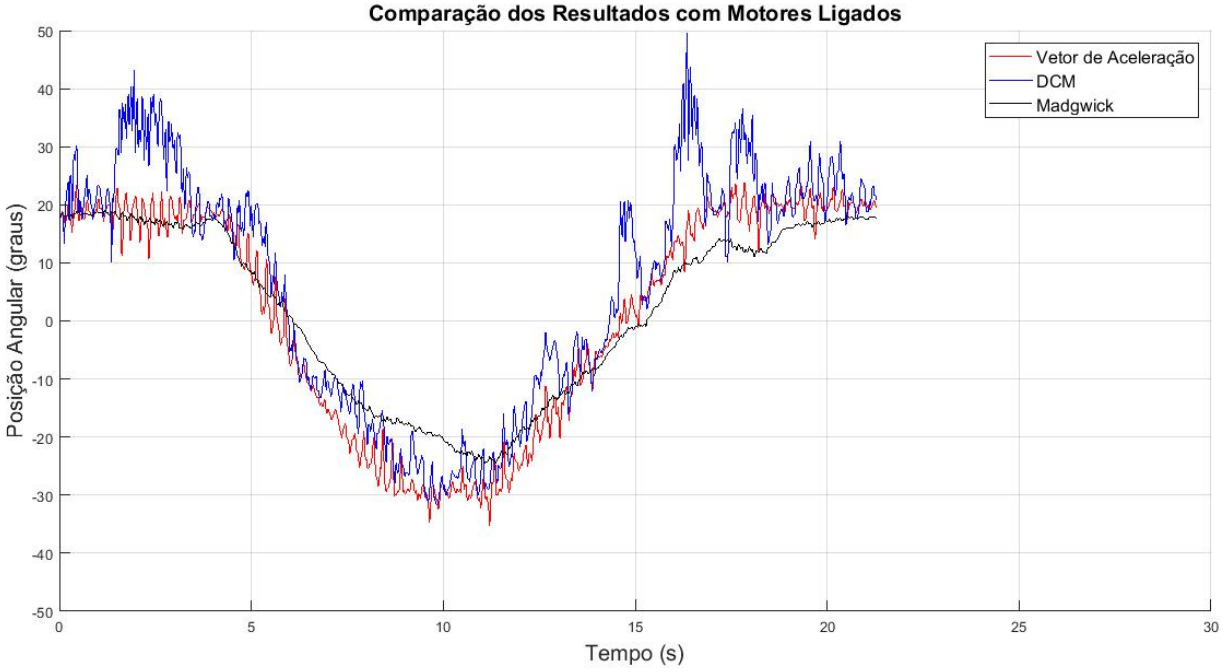
Obteve-se respostas semelhantes quando os motores estão desligados, sendo que todas corresponderam aproximadamente a posição real do eixo durante o teste, como pode ser visto na Figura 40. Já na Figura 41, é possível observar altos níveis de ruído para a resposta do vetor de aceleração e do filtro complementar DCM quando os motores estão ligados. Contudo, o filtro Madgwick rejeitou as vibrações produzidas pelos atuadores, permanecendo apenas uma pequena distorção no resultado da medição. Conclui-se, a partir dos resultados obtidos, que a aplicação do filtro Madgwick é a mais eficaz para ser utilizada na estimação dos ângulos de rotação neste projeto.

Figura 40 - Comparação dos resultados com motores desligados



Fonte: Autoria Própria

Figura 41 - Comparação dos resultados com motores ligados

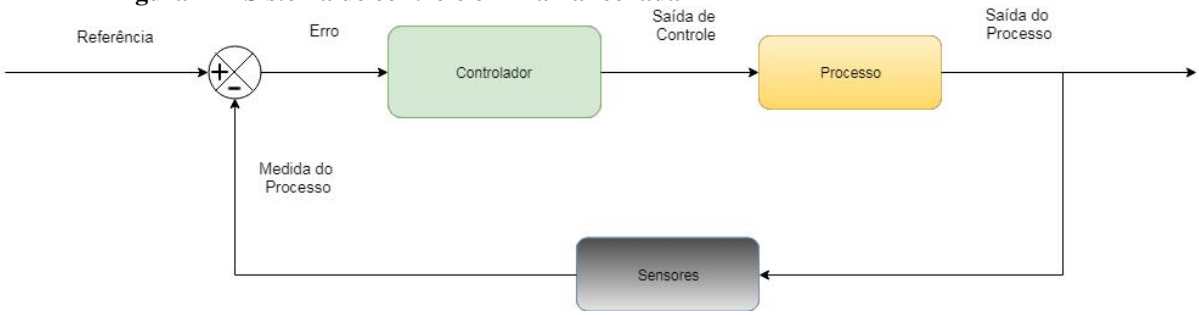


Fonte: Autoria Própria

4.5 CONTROLADOR PROPORCIONAL INTEGRAL DERIVATIVO

Os processos, em sentido amplo, podem ser definidos como uma variável de entrada determinando uma variável de saída, mesmo sem conhecer características internas do processo ou a função de transferência. Em uma planta a ser controlada, é necessário conhecer a variável de saída do sistema por meio de sensores, e comparar tal variável com a referência a ser seguida para obter o erro. Esse requisito é fundamental para compor um sistema em malha fechada, representado na Figura 42. Assim, o erro gerado estará influenciando na atuação do controlador.

Figura 42 - Sistema de controle em malha fechada



Fonte: Baseado em Electrical Technology [58]

Neste formato de sistema é possível corrigir a variável da saída do processo para um valor desejado. Intuitivamente para isso, o controlador precisa adicionar a entrada do processo um valor diretamente proporcional ao erro instantâneo, sendo ponderado pelo valor do ganho proporcional . Ao longo do tempo de execução do processo pode ser acumulado um erro sistemático, configurando uma integral. O ganho integral é multiplicado pela integral do erro para o sistema seguir o valor de referência e minimizar o erro instantâneo. Através da diferença entre o erro atual e o erro imediatamente anterior, obtêm-se a taxa de variação do erro e consequentemente a predição do comportamento do processo. Assim, o adequado ajuste do ganho derivativo possibilita estabilizar o sistema, bem como aprimorar a resposta transitória. [59]

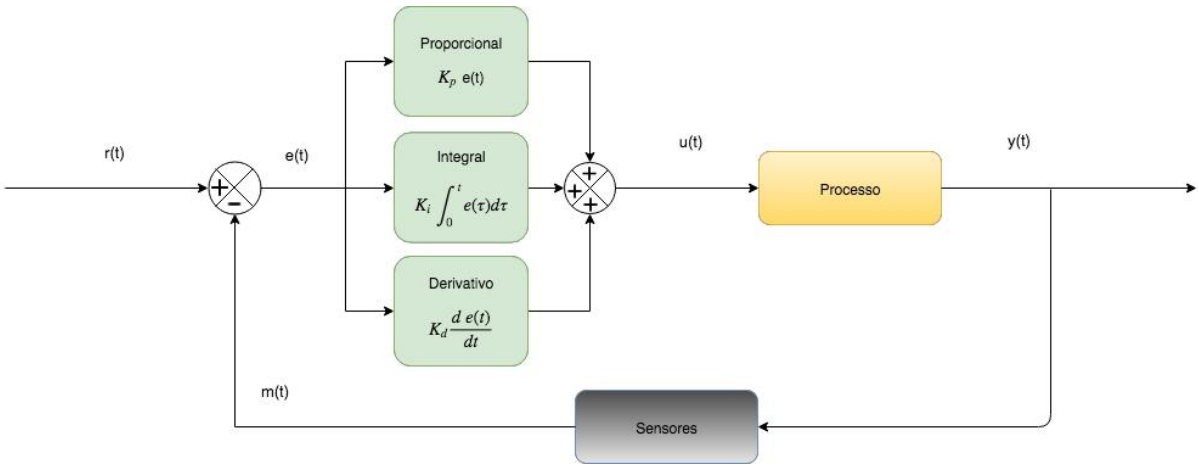
Assim, o controle Proporcional Integral Derivativo (PID) é amplamente utilizado por conta de sua simplicidade. Para implementar essa técnica não é necessário conhecer o modelo da planta, pois os ganhos podem ser ajustados diretamente analisando o comportamento do sistema em malha fechada [60]. A equação do controlador é genérica, mas os três parâmetros de sintonia dependem da natureza específica do sistema. No diagrama da Figura 43 é apresentado o formato básico do sistema de controle PID completo em malha fechada, sendo

r(t) a referência, e(t) o erro, m(t) a medida do processo, u(t) a saída do controlador e y(t) a saída do processo. As equações que definem o controlador PID podem ser vistas nas equações (6) e (7).

$$e(t) = r(t) - m(t) \tag{6}$$

$$u(t) = Kp\, e(t) + Ki \int_0^t e(\tau)\, d\tau + Kd \frac{de(t)}{dt} \tag{7}$$

Figura 43 - Sistema de controle PID



Fonte: **Adaptado de Oscar Liang [61]**

Segundo [62], praticamente todos os controladores fabricados atualmente são baseados em microprocessadores, propriedade presente neste trabalho. Assim, as equações continuas no tempo são aproximadas para equações semelhantes com variáveis discretas no tempo [63], como expressam as equações (8) e (9).

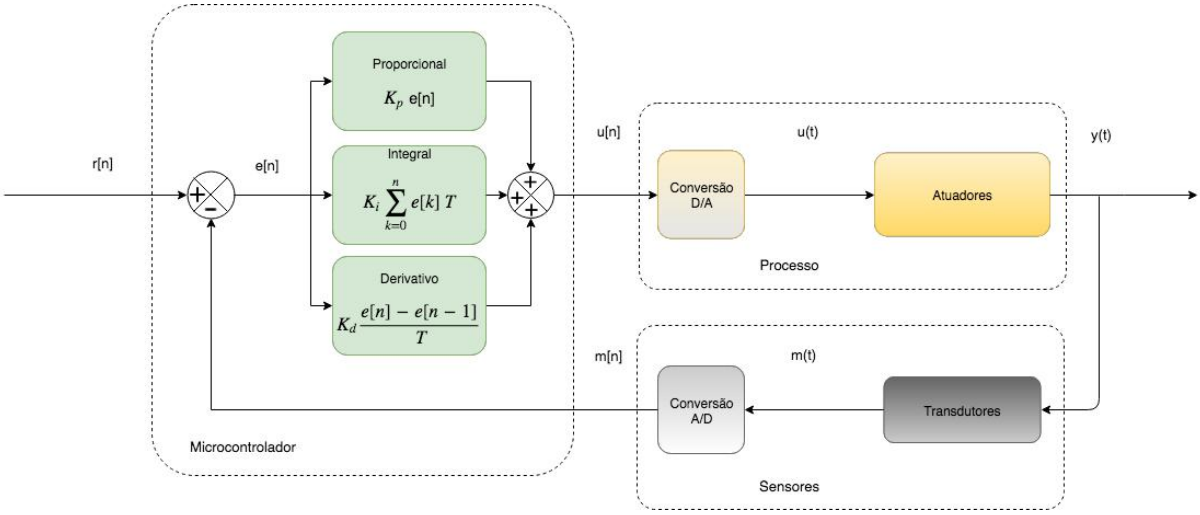
$$e[n] = r[n] - m[n] \tag{8}$$

$$u[n] = K_p\, e[n] + K_i \sum_{k=0}^{n} e[k]\, T + K_d \frac{e[n] - e[n-1]}{T} \tag{9}$$

Entre o controlador digital e os sinais analógicos do processo existem conversores. São eles que fazem a interface entre a entrada e saída do microcontrolador com os sensores e atuadores, respectivamente. A leitura realizada pelos sensores é amostrada no conversor

analógico-digital (A/D) e a resposta do controlador é extrapolada no conversor digital-analógico (D/A) [64]. Esse sistemas com dados quantizados está representado na Figura 44, onde a referência r[n] possui um valor discreto, a medida do sensor m[n] é a amostragem da variável de saída do sistema y(t) e a saída do controlador u[n] é digital.

Figura 44 - Controlador PID com variáveis discretas no tempo



Fonte: Autoria própria

4.5.1 Controle de Posição Angular em Um Eixo

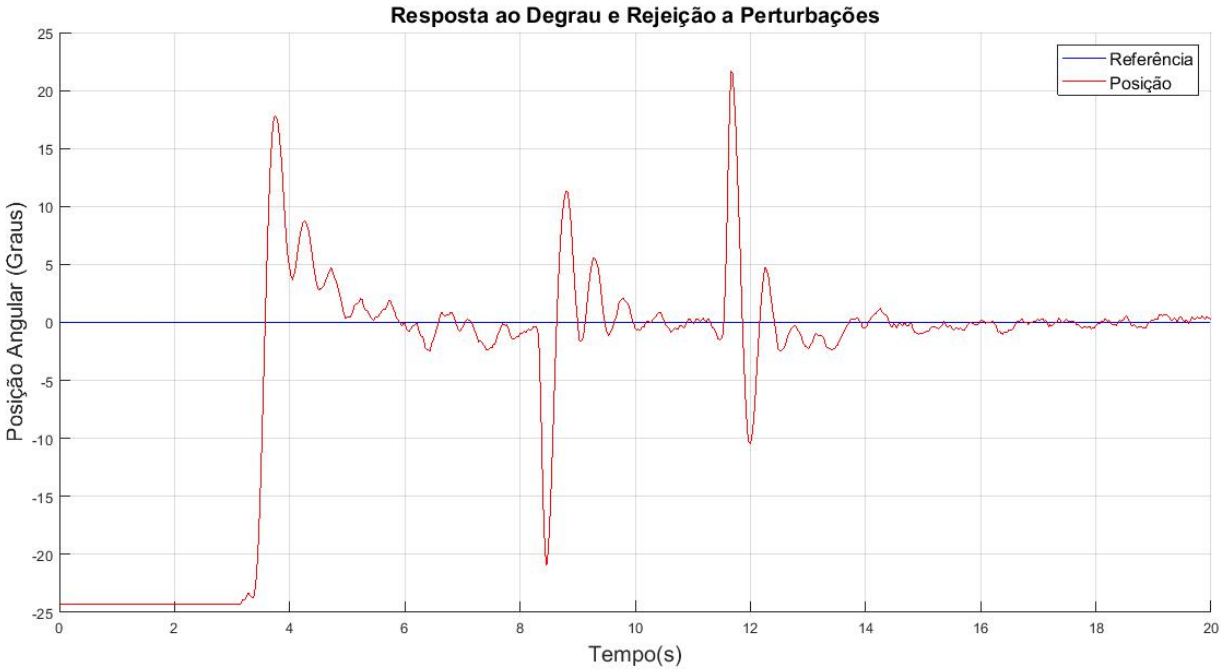
A interação dos elementos no sistema tem a função de controlar a posição angular do eixo de rolamento. Os motores instalados em extremidades opostas são inicialmente acionados com 50% da potência total. Com base no ângulo de referência e no ângulo medido, o sistema de controle produz na sua saída uma diferença de potência, à ser empregada pelos motores, para corrigir o erro calculado.

A sintonia dos ganhos do controlador é realizada nas seguintes etapas: o ganho proporcional foi incrementado até a oscilação do sistema tornar-se constante, na sequência o valor do ganho derivativo passou a ser incrementado e o ganho proporcional reduzido até obter razoável estabilidade. Na etapa seguinte, o ganho integral foi aumentado gradualmente e os outros ganhos regulados para compensar. Então um ajuste fino foi realizado até resultar nos seguintes valores: $= 0{,}986; = 1{,}269; = 0{,}191$.

Na Figura 45, pode ser visto no início do teste que a resposta ao degrau teve aproximadamente 73% de sobressinal, tempo de assentamento de aproximadamente 1,7 s e um erro estacionário de aproximadamente ±2,4°. Após a posição angular estabilizar foram inseridas

perturbações no extremo da armação em aproximadamente 8 s e 11 s, para verificar a capacidade do controlador de manter a estabilidade. Em ambos os casos a estabilidade foi retomada após aproximadamente 1 s. Estes índices validam a metodologia de controle, bem como todas as configurações dos sensores e a estratégia de condicionamento dos sinais.

Figura 45 - Resposta ao degrau e rejeição a perturbações



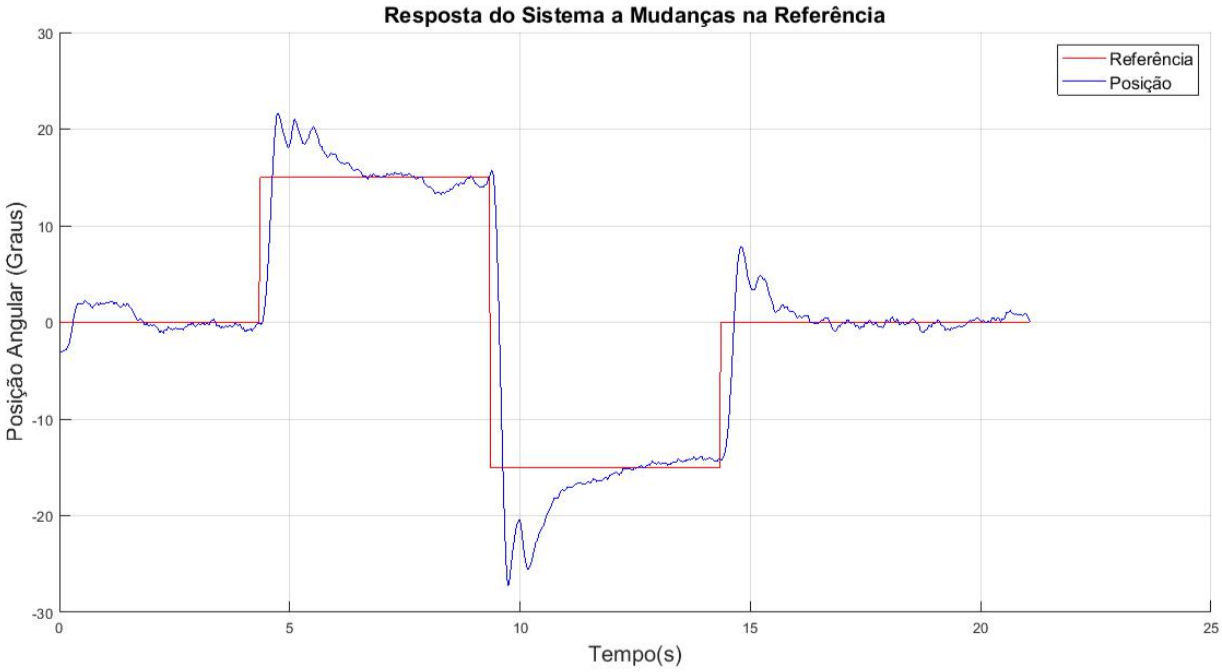
Fonte: Autoria Própria

O gráfico da Figura 46, demonstra a capacidade do controlador de modificar a posição angular conforme é determinado pelo sinal de referência. O procedimento inicia-se em posição angular neutra; em seguida, a referência é colocada em 15° por aproximadamente 5 segundos; então a referência é movida na direção oposta em -15° por aproximadamente 5 segundos; por último, a referência do sistema retorna à posição neutra até o teste ser encerrado.

Nas alterações da referência foi possível levantar os dados de desempenho, com o tempo de assentamento em aproximadamente 2,3 s, sobressinal em torno de 44% e erro em regime estacionário de aproximadamente 0,7°.

Os resultados experimentais demonstram a capacidade de o sistema realizar mudanças abruptas de posição através da aplicação simples e direta da técnica de controle clássica PID.

Figura 46 - Resposta do sistema a mudanças na referência



Fonte: Autoria Própria

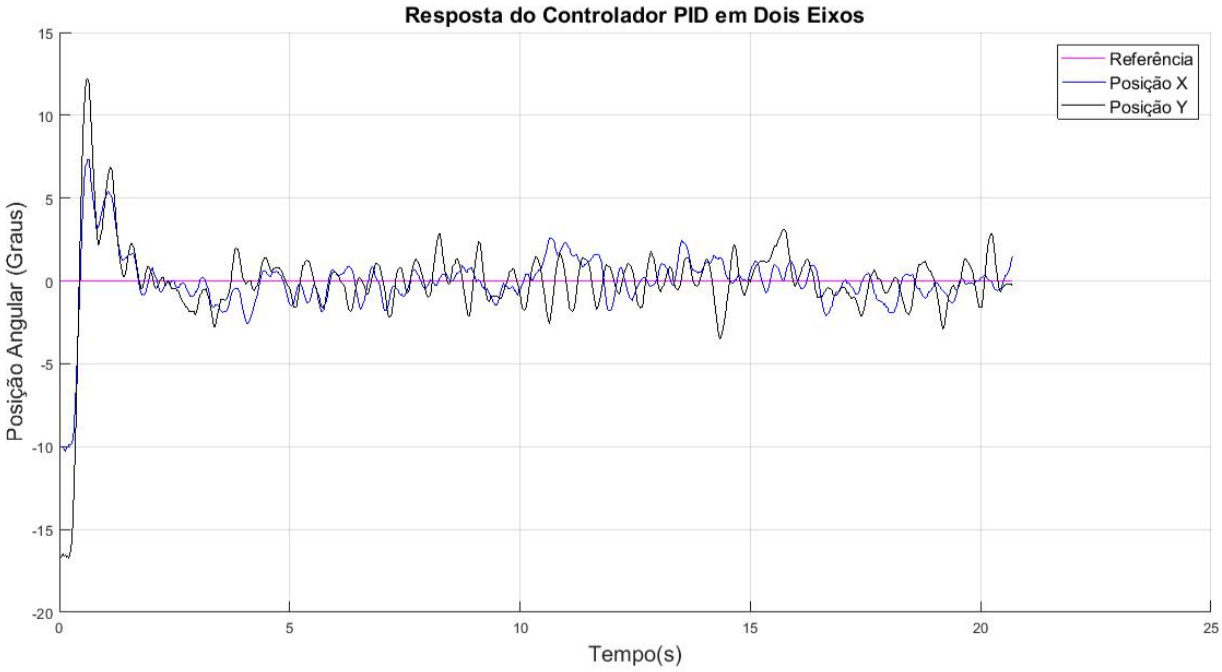
4.5.2 Controle de Posição Angular em Dois Eixos

Os movimentos de rotação do quadricóptero em torno de seu centro de massa determinam os movimentos de translação. Portanto será avaliada a rotação do quadricóptero em torno dos eixos de arfagem e rolamento para subjetivamente avaliar os movimentos de translação.

A estratégia de controle utilizada para controlar apenas o eixo de rolamento na seção 4.5.1, é duplicada para realizar o controle do eixo de arfagem, e assim, realizar um teste simultâneo nos dois eixos horizontais do quadricóptero.

No experimento de resposta ao degrau deste protótipo os eixos iniciaram na posição de repouso da plataforma em ângulos diferentes, aproximadamente -10° no eixo X (em azul) e -16° no eixo Y (em preto). A partir do momento que o sistema de controle passa a agir, ambos os eixos se aproximaram do sinal de referência (0°) com aproximadamente 45% de sobressinal, tempo de assentamento de aproximadamente 1,5 s e erro estacionário de ± 2,0°, como pode ser visto na Figura 47. Adicionalmente, pode-se constatar que a ação simultânea entre os eixos produz um o sobressinal de 45%, que é menor quando comparado ao teste de um eixo isolado, 73%.

Figura 47 - Resposta do controlador PID em dois eixos



Fonte: Autoria Própria

5 CONCLUSÕES

O objeto de estudo deste trabalho, o quadrotor, apresentou características valiosas por ser um veículo aéreo não tripulado de pequeno porte e possuir diversos dispositivos eletrônicos. Dentre eles o receptor de rádio frequência, acelerômetro, giroscópio, magnetômetro, controlador eletrônico de velocidade e finalmente, o microcontrolador, responsável pela integração dos demais no sistema.

Trata-se, portanto, de um sistema eletrônico embarcado com sinais de sensoriamento e componentes de potência. Diversas técnicas, algoritmos e métodos puderam ser testados por conta da adaptabilidade de um protótipo. Assim, a cada etapa novos conhecimentos foram adquiridos em ciclos progressivos de desenvolvimento, planejamento e implementação.

O condicionamento dos sinais digitais dos sensores foi uma etapa crítica da pesquisa, pois os elevados níveis de ruído intrínseco dos sensores precisaram ser contornados. Foram utilizados filtros do próprio giroscópio com resultados positivos e na fusão dos sensores a aplicação do filtro complementar DCM teve resultados aceitáveis nos testes iniciais, entretanto com a ativação dos motores, as medidas divergiram da posição física. Na sequência, a aplicação do filtro Madgwick obteve os resultados suficientes para esse trabalho.

Por fim, uma abordagem simplificada de controle, duas malhas fechadas independentes, foram capazes de realizar a estabilização do quadrotor. Assim, em mais um caso, a técnica clássica de controle PID atendeu as necessidades de aplicação.

Visando complementar o trabalho aqui apresentado, algumas propostas para projetos futuros podem ser consideradas.

O sensoriamento pode ser estudado separadamente para que os sistemas de posicionamento inercial sejam aprimorados. As opções sugeridas são: utilizar maiores frequências de amostragem, comparar a medição dos sensores com a posição real, utilizar múltiplos sensores para aumentar a confiabilidade e avaliar os métodos de fusão de sensores.

É possível ainda, explorar outros paradigmas de controle com conceitos avançados, usar técnicas de controle não linear ou lógica nebulosa (*fuzzy*). Para garantir a robustez do controlador podem ser implementados artifícios de compensação a falhas.

Além disso, deve ser considerada a construção de plataformas de testes que possibilitem ensaios com todos os graus de liberdade, permitindo expandir a avaliação do quadricóptero. Em todos os casos, a programação do microcontrolador é fundamental, portanto abordar estruturas de programação avançadas é importante para o progresso dos sistemas embarcados.

6 REFERÊNCIAS BIBLIOGRÁFICAS

[1] ARDUPILOT, Dev Team. **What is a Multicopter and How Does it Work?**. Disponível em: <http://ardupilot.org/copter/docs/what-is-a-multicopter-and-how-does-it-work.html> Acesso em: 12 set 2016.

[2] QUEIROZ, Juliano R. **Construção de um veículo aéreo não tripulado na configuração quadrirrotor como plataforma de estudos**. 2014. 115 folhas. Trabalho de Conclusão de Curso 2 (Bacharelado em Engenharia Eletrônica) - Universidade Tecnológica Federal do Paraná. Toledo, 2014.

[3] LEISHMAN, J. G. **Principles of helicopter aerodynamics**. Cambridge University Press. New York, 2000.

[4] HIRSCHBERG, Michael J. **The American Helicopter: An Overview of Helicopter Developments in America 1908-1999**. Disponível em: <http://www.iasa.com.au/folders/Publications/pdf_library/ospreypdfs/The%20American%20Helicopter.pdf>. Acesso em: 5 set 2018.

[5] SOUZA, Marisabel P. **Implementação de um Simulador de Robôs Quadrotores Para a Construção Civil em VRML**. 2013. Monografia (Bacharel em Engenharia Elétrica). Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense. Pelotas, Brasil. 2013.

[6] VINHOLES, Thiago. **O Avião e Sua Múltipla Paternidade**. Disponível em: <http://airway.uol.com.br/o-aviao-e-sua-multipla-paternidade> Acesso em: 2 nov 2016.

[7] Imagem disponível em: <http://fr.wikipedia.org/wiki/Quadrirotor#mediaviewer/File:Oemichen2.jpg> Acesso em: 16 nov 2014.

[8] Imagem disponível em: <http://pt.wikipedia.org/wiki/Quadrotor#mediaviewer/File:De_Bothezat_Quadrotor.jpg> Acesso em: 16 nov 2014.

[9] AMAZON. **Amazon Prime Air.** Disponível em: <http://www.amazon.com/b?node=8037720011> Acesso em: 16 nov 2014.

[10] KROO, I et al. **The mesicopter: a miniature rotorcraft concept, phase ii final report**. Relatório técnico, Stanford University, 2001.

[11] HOFFMAN, G et al. **The stanford test bed of autonomous rotorcraft for multi agente control (Starmac)**. The 23rd digital avionics systems conference, v. 2, p. 12.E.4-10, Salt Lake City, 2004.

[12] BOUABDALLAH, Samir. **Design and control of quadrotors with application to autonomous flying**. 2007. 129 f. Tese (Doutorado em Ciências) - Faculté des Sciences et Techniques de L'ingénieur, École Polytechnique Fédérale de Lausanne, Lausanne, 2007.

[13] BOUABDALLAH, S.; MURRIERI, P.; SIEGWART, R. Design and control of an indoor quadrotor. IEEE International Conference on Robotics and Automation, New Orleans, p. 4393-4398, abril 2004.

[14] BOUABDALLAH, S.; NOTH, A.; SIEGWART, R. PID vs LQ control techniques applied to an indoor micro quadrotor. IEEE International Conference on Robotics and Automation, Sendal, p. 2451-2456, out. 2004.

[15] BOUABDALLAH, S.; SIEGWART, R. 2005. **Backstepping and sliding-mode techniques applied to an indoor micro quadrotor**. In Proceedings of the 2005 IEEE International Conference on Robotics and Automation, Barcelona, Spain, 2005.

[16] Office of the Secretary of Defence. **Unmanned Aircraft Systems Roadmap: 2005-2030**, 2008.

[17] DECEA. **Voos de RPAS (drones). Entenda a nova legislação do DECEA**. Disponível em: <http://www.decea.gov.br/?i=midia-e-informacao&p=pg_noticia&materia=voos-de-rpas-drones-entenda-a-nova-legislacao-do-decea> Acesso em: 2 nov 2016.

[18] KIM, S. K.; TILBURY, D. M. . **Mathematical modeling and experimental identification of a model helicopter**. AIAA Modeling and Simulation Technologies Conference and Exhibit, Boston, MA, 1998.

[19] CASTILLO, GARCIA et al. "**Modelado y estabilización de un helicóptero con cuatro rotores**." Revista Iberoamericana de Automática e Informática Industrial RIAI, 2007.

[20] Imagem disponível em: <http://upload.wikimedia.org/wikipedia/commons/7/79/Aeryon_Scout_In_Flight.jpg> Acesso em: 1 nov 2014.

[21] Imagem disponível em: <https://upload.wikimedia.org/wikipedia/commons/thumb/d/db/Axis.png/200px-Axis.png> Acesso em: 9 nov 2016.

[22] COSTA, Exuperry B. **Algoritmos de Controle Aplicados à Estabilização do Vôo de um Quadrotor**. 2012. Dissertação (Mestrado em Engenharia Elétrica) - Universidade Federal de Juiz de Fora. Juiz de Fora, 2012.

[23] SÁ, Rejane C. **Construção, modelagem dinâmica e controle PID para estabilidade de um veículo aéreo não tripulado do tipo quadrirotor**. Dissertação (Mestrado em Engenharia de Teleinformática) - Centro de Tecnologia, Universidade Federal do Ceará.

[24] RIBEIRO, Marcos S. **Construindo um quadricóptero.** Disponível em: <http://www.quad.marksr.com.br/p/guia-basico-parte-1.html> Acesso em: 2 nov 2014.

[25] Imagem disponível em: <https://hobbyking.com/media/catalog/product/cache/1/image/565x414/9df78eab33525d08d6e5fb8d27136e95/legacy/catalog/22397.jpg > Acesso em: 9 nov 2016

[26] VIEIRA, J. C. S. **Plataforma móvel aérea quadrotor**. 2011. Dissertação (Mestrado em Engenharia Eletrônica Industrial e Computadores) - Escola de Engenharia, Universidade do Minho, Guimarães, 2011.

[27] MADGWICK, Sebastian O. H. **An efficient orientation filter for inertial and inertial/magnetic sensor arrays**. University of Bristol, 2010. Disponível em: <http://x-io.co.uk/res/doc/madgwick_internal_report.pdf> Acesso em: 25 mai 2016.

[28] MAGNUSSEN, Oyvind. SKJOHAUG, Kjell E. **Modeling, design and experimental study for a quadcopter system construction**. Dissertação (Mestrado em Engenharia) – Faculty of Technology and Science, University of Agder, Grimstad, 2011.

[29] ANALOG DEVICES. **ADXL345 Digital Accelerometer Datasheet.** Rev. D. Norwood, 2013.

[30] FEI, J. **Robust adaptive vibration tracking control for a micro-electro-mechanical systems vibratory gyroscope with bound estimation.** IET control theory & applications 4, no. 6, p. 1019-1026, 2010.

[31] STMICROELECTRONICS. **L3G44200D MEMS Motion Sensor: Ultra-Stable Three-Axis Digital Output Gyroscope**. Rev. 3. 2010.

[32] HONEYWELL. **3-Axis Digital Compass IC HMC5883L**. Rev. E. Plymounth, 2013.

[33] TEXAS INSTRUMENTS. Tiva™ C Series ARM® Cortex™-M Microcontrollers Disponível em: <http://www.ti.com/lit/ml/spmt284/spmt284.pdf> Acesso em: 8 set 2018.

[34] TEXAS INSTRUMENTS. Tiva™ TM4C1233H6PM Microcontroller Datasheet. Disponível em: <http://www.ti.com/lit/ds/symlink/tm4c1233h6pm.pdf> Acesso em: 8 set 2018.

[35] AFONSO, Luís R. S. **Tiva Tutorials**. Disponível em: <https://sites.google.com/site/luiselectronicprojects/tutorials/tiva-tutorials> Acesso em: 2 nov 2016.

[36] Texas Instruments. **Getting Started with the TIVA™ C Series TM4C123G LaunchPad**. Disponível em: <http://processors.wiki.ti.com/index.php/Getting_Started_with_the_TIVA%E2%84%A2_C_Series_TM4C123G_LaunchPad> Acesso em: 2 nov 2016.

[37] Texas Instruments. **TivaWare™ Peripheral Driver Library**. 2013. Disponível em: <http://www.ti.com/lit/ug/spmu298d/spmu298d.pdf> Acesso em: 10 nov 2016.

[38] Texas Instruments. **TivaWare™ Sensor Library**. 2015. Disponível em: <http://www.ti.com/lit/ug/spmu371d/spmu371d.pdf> Acesso em: 10 nov 2016.

[39] Social Ledge. **S14: Quadcopter**. Dísponível em: <http://www.socialledge.com/sjsu/index.php?title=S14:_Quadcopter> Acesso em 18 out 2016.

[40] Texas Instruments. **TivaWare™ Utilities Library**. 2013. Disponível em: <https://e2e.ti.com/cfs-file/__key/telligent-evolution-components-attachments/00-471-01-00-01-04-33-05/SW_2D00_TM4C_2D00_UTILS_2D00_UG_2D00_2.0.pdf> Acesso em: 8 abr 2017.

[41] FrSky RC. **Instruction Manual for FrSky D8R-XP**. Dísponível em: <https://hobbyking.com/media/file/61219670X12918X30.pdf> Acesso em: 11 jun 2017.

[42] PIHLAJAMAA, Joonas. **Stellaris Launchpad PWM Tutorial**. Disponível em: <http://codeandlife.com/2012/10/30/stellaris-launchpad-pwm-tutorial/>. Acesso em 18 mar 2015.

[43] BARR, Michael. **Introduction to Pulse Width Modulation**. Disponível em: <https://www.embedded.com/electronics-blogs/beginner-s-corner/4023833/Introduction-to-Pulse-Width-Modulation>. Acesso em 1 mar 2017.

[44] CARVALHO, A. et al. **Experimental Comparacion of Sensor Fusion Algorithms for Attitude Estimation**. Dipartimento di Ingegneria Industriale e dell'Informazione, Seconda

[45] BALUTA, Sergiu. **A Guide to Using IMU (Accelerometer and Gyroscope Devices) in Embedded Applications**. Starlino Electronics Project. 2009. Disponível em: <http://www.starlino.com/imu_guide.html> Acesso em: 10 nov 2016.

[46] KONVALIN, Christopher. **MEMSense Technical Document Compensating for Tilt, Hard Iron and Soft Iron Effects**. Revison 1.2. 2008. Disponível em: <http://www.ckdevices.com/datasheets/MTD-0802_1.2_Magnetometer_Calibration.pdf> Acesso em: 17 ago 2016.

[47] WINER, Kris. **Simple and Effective Magnetometer Calibration**. 2014. Disponível em: <https://github.com/kriswiner/MPU-6050/wiki/Simple-and-Effective-Magnetometer-Calibration> Acesso em: 4 out 2016.

[48] KALMAN, Rudolph E. **A new approach to linear filtering and prediction problems**. Journal of basic Engineering 82.1, 1960.

[49] BARSHAN, B.; DURRANT-WHYTE, H. F. **Inertial navigation systems for mobile robots.** 1995.

[50] MARINS, J. L. et al. **An Extended Kalman Filter for Quaternion-Based Orientation Estimation Using MARG Sensors**. International Conference on Intelligent Robots and Systems. 2001.

[51] BALUTA, Sergiu. **DCM Tutorial - An Introduction to Orientation Kinematics**. Starlino Electronics Project. 2011. Disponível em: <http://www.starlino.com/dcm_tutorial.html> Acesso em: 10 nov 2016.

[52] PREMERLANI, Willian; BIZARD, Paul. **Direction Cosine Matrix IMU: Theory**. 2009

[53] MAHONY, Robert et al. **Complementary Filter Design on the Special Orthogonal Group SO(3)**. 44th IEEE Conference on Decision and Control and the European Control Conference 2005. Seville, 2005

[54] EDVARDSEN Michael; RIETZ, Joel. **Sensor-Based Intelligent Positioning and Monitoring System**. Master Thesis - Uppsala Universitet, Uppsala, 2017.

[55] CH ROBOTICS. **Understanding Quaternions**. Disponível em: <http://www.chrobotics.com/library/understanding-quaternions> Acesso em: 9 set 2016.

[56] X-IO TECHNOLOGIES. **Open Source IMU and AHRS Algorithms**. Disponível em: <http://x-io.co.uk/open-source-imu-and-ahrs-algorithms/> Acesso em: 11 nov 2016

[57] WINER, Kris. **GY-80 Basic AHRS**. Disponível em: <https://github.com/kriswiner/GY-80/blob/master/GY80BasicAHRS.ino> Acesso em: 9 set 2016.

[58] ELECTRICAL TECHNOLOGY. **What is PID Controller and How it Works**. Disponível em: <https://www.electricaltechnology.org/2015/10/what-is-pid-controller-how-it-works.html> Acesso em: 14 nov 2017.

[59] GUERRA, Leonardo N. A. **Uso de Compensador PID no Controle da Taxa de Variação de Temperatura em um Forno Elétrico a Resistência**. 2006. Projeto de Graduação (Bacharelado em Engenharia Elétrica). Universidade Federal do Rio de Janeiro. Rio de Janeiro, 2006.

[60] NATIONAL INSTRUMENTS. **Explicando a Teoria PID**. 2011. Disponível em: <http://www.ni.com/white-paper/3782/pt/>. Acesso em: 9 set 2016.

[61] LIANG, Oscar. **Quadcopter PID Explained**. Disponível em: <https://oscarliang.com/quadcopter-pid-explained-tuning/>. Acesso em: 2 mar 2017.

[62] ÅSTRÖM, Karl J., HÄGGLUND, Tore. **PID controllers: theory, design, and tuning**. Vol. 2. Research Triangle Park: Instrument Society of America, 1995.

[63] LEIGH, James R. **Applied Control Theory**. 2.ed. London: IET, 1987.

[64] NISE, Norman S. **Engenharia de Sistemas de Controle**. 3.ed. Rio de Janeiro: LTC, 2002.

APÊNDICE A - CÓDIGO FONTE

```
#include <stdint.h>
#include <stdbool.h>
#include "stdlib.h"
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_uart.h"
#include "inc/hw_gpio.h"
#include "inc/hw_pwm.h"
#include "inc/hw_types.h"
#include "inc/hw_i2c.h"
#include "driverlib/timer.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "driverlib/udma.h"
#include "driverlib/pwm.h"
#include "driverlib/ssi.h"
#include "driverlib/systick.h"
#include "driverlib/i2c.h"
#include "driverlib/fpu.h"
#include "utils/uartstdio.h"
#include "sensorlib/comp_dcm.h"
#include <math.h>
#include <string.h>

//*****************************************************************************
//Define of macros

#define INITIAL 0
#define VERIFY_ACC 1
#define READ_ACC 2
#define PROCESS_ACC 3
#define VERIFY_GYRO 4
#define READ_GYRO 5
#define PROCESS_GYRO 6
#define VERIFY_MAG 7
#define READ_MAG 8
#define PROCESS_MAG 9
#define FILTER_UPDATE 10
#define PID_UPDATE 11
#define SEND_SERIAL 12

#define NUM_TASKS 13

typedef enum
{
	BLOCKED, READY, DONE
} TaskStatus;

typedef struct
{
	TaskStatus tdef_Tasks_Status[NUM_TASKS];
```

```
	uint32_t pui32_Tasks_Start_Time[NUM_TASKS];
	uint32_t pui32_Tasks_Finish_Time[NUM_TASKS];
} TM;

TM TM_Schedule =
{
		{ BLOCKED, BLOCKED, BLOCKED, BLOCKED, BLOCKED, BLOCKED,
		BLOCKED, BLOCKED, BLOCKED, BLOCKED, BLOCKED, BLOCKED, BLOCKED },
		{0,0,0,0,0,0,0,0,0,0,0,0,0},
		{0,0,0,0,0,0,0,0,0,0,0,0,0}
};

#ifndef M_PI
#define M_PI 3.1415926f
#endif

#define SAMPLING_FREQUENCY 390.0f

#define def_CONVERT_RADIUS_TO_DEGRESS 180.0f / M_PI

#define ACCEL_SLAVE_ADDR 0x53		// I2C Slave address
#define GYRO_SLAVE_ADDR 0x69		//slave_addr = 1101001b i.e. Tiva swap the
address
#define MAG_SLAVE_ADDR 0x1E		//incremented by 1 automatically after
readed

#define def_ACC_OFFSET_X -11
#define def_ACC_OFFSET_Y -4
#define def_ACC_OFFSET_Z -6

#define def_ACC_SCALE_FACTOR_X 0.03845745f
#define def_ACC_SCALE_FACTOR_Y 0.03891528f
#define def_ACC_SCALE_FACTOR_Z 0.03954294f

#define def_ACC_NUM_SAMPLES 20

#define def_MAG_OFFSET_X 100
#define def_MAG_OFFSET_Y -99
#define def_MAG_OFFSET_Z -71

#define def_MAG_SCALE_FACTOR_X 0.00092000f //=0.92 * 0.001 * (628/610) -> result
in Gauss
#define def_MAG_SCALE_FACTOR_Y 0.00092000f //=0.92 * 0.001 * (628/635) -> result
in Gauss
#define def_MAG_SCALE_FACTOR_Z 0.00092000f //=0.92 * 0.001 * (628/628) -> result
in Gauss

#define ANG_RANGE 30.0f

#define PWM_RANGE 1024.0f

#define PWM_MAX 1950
#define PWM_STANDBY 988
#define PWM_MIN 1050 //determined through tests #define PWM_HALF 1500


//*****************************************************************************
// Global Instance structure to manage the DCM state.
```

```
tCompDCM g_sCompDCMInst;

float beta = 0.4f;//worked at 0.04f //from Madgwick //2.5f to 10 first seconds

//float beta = 0.6046f; //from Kriswiner

float q0 = 1.0f, q1 = 0.0f, q2 = 0.0f, q3 = 0.0f;// quaternion of sensor frame
relative to auxiliary frame

//*****************************************************************************
//Global variables

uint8_t gui8_First_Timeout_Flag = 1;
uint8_t gui8_Fisrt_Reading_Flag = 1;
uint32_t ui32_IntStatus = 0;

uint16_t gui16_AuxTick = 0;
uint8_t gui8_Second = 0;

//*****************************************************************************
//Sensor Global variables

uint8_t gpui8_Buffer[6];  //Buffer of multiple data from I2C

//*****************************************************************************
//Radio Global variables

//Stores pulse length
//////////////////////////////////////////why uint32? It could be uint16? Could not

uint32_t gui32_Radio_PulseE2 = PWM_STANDBY;     //Chanel 1, Enable motors
uint32_t gui32_Radio_PulseE3 = PWM_STANDBY;     //Chanel 2, Potentiometer 3

uint32_t gui32_Radio_PulseA2 = PWM_STANDBY;     //Conected to Chanel 3, that means
it's the Elevator
uint32_t gui32_Radio_PulseA3 = PWM_STANDBY;     //Chanel 4, Aileron
uint32_t gui32_Radio_PulseA4 = PWM_STANDBY;     //Chanel 5, Throtle
uint32_t gui32_Radio_PulseA5 = PWM_STANDBY;     //Chanel 6, Rudder //Chanel
uint32_t gui32_Radio_PulseA6 = PWM_STANDBY;     7, Potentiometer 1 //Chanel
uint32_t gui32_Radio_PulseA7 = PWM_STANDBY;     8, Potentiometer 2

//float_t test = 0.0f;

//*****************************************************************************
//Functions declarations

void InitLedSWs(void);

void InitTimer0(void);
void Timer0IntHandler(void);

void InitWTimer0PWM(void);
void InitWTimer1PWM(void);

void InitUART0(void);

void InitSysTick(void);
void SycTickInt(void);
```

```
void InitRadioInput();
void PortAIntHandler();
void PortEIntHandler();

//*****************************************************************************

void InitI2C0(void);
void I2CSendSingleData(uint8_t ui8SlaveAddr, uint8_t ui8RegAddr, uint8_t ui8Data);
uint32_t I2CReceiveSingleData(uint8_t ui8SlaveAddr, uint8_t ui8RegAddr);
void I2CReceiveMultipleData(uint8_t ui8SlaveAddr, uint8_t ui8FirstReg, uint8_t
ui8NumReg);

void InitADXL345(void);
void InitL3G4200D(void);
void InitHMC588L(void);

void MadgwickAHRSupdate(float gx, float gy, float gz, float ax, float ay, float
az, float mx, float my, float mz);
void MadgwickAHRSupdateIMU(float gx, float gy, float gz, float ax, float ay, float
az);
float invSqrt(float x);

void PrintFloatVariable(float Float_Variable, uint8_t Enter);

//*****************************************************************************

int main(void)

{

//*****************************************************************************
// Set System clock

      // system clock = 80 Mhz
      SysCtlClockSet( SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL     | SYSCTL_XTAL_16MHZ |
SYSCTL_OSC_MAIN);

//    for ( ui8_I = 0; ui8_I < NUM_TASKS; ui8_I++)
//          TM_Schedule.tdef_Tasks_Status[ui8_I] = BLOCKED;

//*****************************************************************************
//General Variables

      uint8_t ui8_Data_Ready = 0;
      uint8_t ui8_I;
//    uint8_t ui8_Cont = 0;

      uint8_t ui8_cycle_count = 0;

//*****************************************************************************
//Acelerometer Variables

      uint8_t pui8_Acc_Read_Data[6];
      int16_t i16_Acc_Raw_Data_X, i16_Acc_Raw_Data_Y,
      i16_Acc_Raw_Data_Z; float f_Acc_X, f_Acc_Y, f_Acc_Z;

/*
      int16_t pi16_Acc_Vector_X[def_ACC_NUM_SAMPLES],
pi16_Acc_Vector_Y[def_ACC_NUM_SAMPLES], pi16_Acc_Vector_Z[def_ACC_NUM_SAMPLES];
```

```
	int32_t i32_Acc_Add_X, i32_Acc_Add_Y, i32_Acc_Add_Z;
	float f_Acc_Avg_X, f_Acc_Avg_Y, f_Acc_Avg_Z; float
	f_Acc_Module;
	float  f_Ang_X, f_Ang_Y, f_Ang_Z;
	*/

//*************************************************************************
//Gyroscope variables

	uint8_t pui8_Gyro_Read_Data[6];
	int16_t i16_Gyro_Raw_X, i16_Gyro_Raw_Y, i16_Gyro_Raw_Z; //integer with
2 bytes: -32768 to 32768
	float f_Gyro_X, f_Gyro_Y, f_Gyro_Z;

//*************************************************************************
//Magnetometer Variables

	uint8_t pui8_Mag_Read_Data[6];
	int16_t i16_Mag_Raw_X, i16_Mag_Raw_Y, i16_Mag_Raw_Z; //integer with 2 bytes:
-32768 to 32768
	float f_Mag_X, f_Mag_Y, f_Mag_Z;

//*************************************************************************
//Fusion Filter Variables

	float f_Euler_X, f_Euler_Y, f_Euler_Z;
	//float f_Euler_X2, f_Euler_Y2, f_Euler_Z2;
	//float f_quat4d_test, heading, attitude, bank, q0t, q1t, q2t,
	q3t; float f_Madgwick_yaw, f_Madgwick_pitch, f_Madgwick_roll;
	//uint32_t Time_Beta = 0;

//*************************************************************************
//PID Variables

	float f_Throtle_Porcentage = 0.0f;
	uint16_t ui16_Enable_Control = PWM_STANDBY;
	float dt = 1.0f / SAMPLING_FREQUENCY;

	float f_Saturation = 0.0f;

	float Kp = 0.986f; //2.52f;
	float Ki = 1.269f; //3.0f;
	float Kd = 0.191f; //0.53f;

	float f_pre_error_Y = 0, f_setpoint_Y = 0, f_integral_Y = 0;
	float f_actual_error_Y, f_actual_position_Y, f_derivative_Y,
	f_output_Y; float f_Output_Left, f_Output_Right;

	float f_pre_error_X = 0, f_setpoint_X = 0, f_integral_X = 0;
	float f_actual_error_X, f_actual_position_X, f_derivative_X,
	f_output_X; float f_Output_Up, f_Output_Down;

	//float Ki_Calc;

//*************************************************************************
//Initialize the peripherals

	InitUART0();
	InitLedSWs();
```

```
	InitI2C0();
	InitADXL345();
	InitL3G4200D();
	InitHMC588L();

	SysCtlDelay(SysCtlClockGet() / 2); // wait 0.5 s for enable all functions
of sensors

//*****************************************************************************
// Initialize the DCM system. 390 hz sample rate.

	// WEIGHT STANDART accel weight = .2, gyro weight = .6, mag weight = .2
	CompDCMInit(&g_sCompDCMInst, 1.0f / SAMPLING_FREQUENCY, 0.10f, 0.80f,
0.10f);

//*****************************************************************************
//Initialize the timers and interruptions

	InitRadioInput();

	//InitSysTick();
	InitWTimer0PWM();
	InitWTimer1PWM();

	InitTimer0();

//*****************************************************************************
//Operational System

	for (;;) //ever
			{
//*****************************************************************************
//Initial Task

		if (TM_Schedule.tdef_Tasks_Status[INITIAL] == READY) {
			TM_Schedule.pui32_Tasks_Start_Time[INITIAL] =
TimerValueGet(TIMER0_BASE, TIMER_A);

			TM_Schedule.tdef_Tasks_Status[VERIFY_ACC] = READY;
			TM_Schedule.tdef_Tasks_Status[READ_ACC] = BLOCKED;
			TM_Schedule.tdef_Tasks_Status[PROCESS_ACC] = BLOCKED;
			TM_Schedule.tdef_Tasks_Status[VERIFY_GYRO] = READY;
			TM_Schedule.tdef_Tasks_Status[READ_GYRO] = BLOCKED;
			TM_Schedule.tdef_Tasks_Status[PROCESS_GYRO] = BLOCKED;
			TM_Schedule.tdef_Tasks_Status[VERIFY_MAG] = READY;
			TM_Schedule.tdef_Tasks_Status[READ_MAG] = BLOCKED;
			TM_Schedule.tdef_Tasks_Status[PROCESS_MAG] = BLOCKED;
			TM_Schedule.tdef_Tasks_Status[FILTER_UPDATE] = BLOCKED;
			TM_Schedule.tdef_Tasks_Status[PID_UPDATE] = BLOCKED;
			TM_Schedule.tdef_Tasks_Status[SEND_SERIAL] = BLOCKED;

			for (ui8_I = 0; ui8_I <= 11; ui8_I++) {
				TM_Schedule.pui32_Tasks_Start_Time[ui8_I] = 0;
				TM_Schedule.pui32_Tasks_Finish_Time[ui8_I] = 0;
			}

			TM_Schedule.tdef_Tasks_Status[INITIAL] = DONE;
```

```
                    TM_Schedule.pui32_Tasks_Finish_Time[INITIAL]
= TimerValueGet(TIMER0_BASE, TIMER_A);

                    //UARTprintf("0");
              }

//*****************************************************************************
//Verify Acelerometer

              if (TM_Schedule.tdef_Tasks_Status[VERIFY_ACC] == READY) {
                    TM_Schedule.pui32_Tasks_Start_Time[VERIFY_ACC] =
TimerValueGet(TIMER0_BASE, TIMER_A);

                    //verify the bit D7-DATA_READY in Register 0x30—INT_SOURCE
                    ui8_Data_Ready = I2CReceiveSingleData(ACCEL_SLAVE_ADDR, 0x30); if
                    ((ui8_Data_Ready & 0b10000000) > 0) //if D7 is 1 then do it
below 0b10000000
                    {
                          TM_Schedule.tdef_Tasks_Status[VERIFY_ACC] = DONE;
                          TM_Schedule.tdef_Tasks_Status[READ_ACC] = READY;
                    }

                    TM_Schedule.pui32_Tasks_Finish_Time[VERIFY_ACC] =
TimerValueGet(TIMER0_BASE, TIMER_A);
                    //UARTprintf("1");
              }
//*****************************************************************************
//Read Acelerometer

              if (TM_Schedule.tdef_Tasks_Status[READ_ACC] == READY) {
                    TM_Schedule.pui32_Tasks_Start_Time[READ_ACC] =
TimerValueGet(TIMER0_BASE, TIMER_A);

                    //Routine to read accelerometer data by I2C bus
                    I2CReceiveMultipleData(ACCEL_SLAVE_ADDR, 0x32, 6);

                    // Routine to atribute buffer of I2C learning
                    for (ui8_I = 0; ui8_I < 6; ui8_I++) {
                          pui8_Acc_Read_Data[ui8_I] = gpui8_Buffer[ui8_I];

                    }

                    TM_Schedule.tdef_Tasks_Status[READ_ACC] = DONE;
                    TM_Schedule.tdef_Tasks_Status[PROCESS_ACC] = READY;

                    TM_Schedule.pui32_Tasks_Finish_Time[READ_ACC] =
TimerValueGet(TIMER0_BASE, TIMER_A);
                    //UARTprintf("2");
              }
//*****************************************************************************
//Process Acelerometer

              if (TM_Schedule.tdef_Tasks_Status[PROCESS_ACC] == READY) {
                    TM_Schedule.pui32_Tasks_Start_Time[PROCESS_ACC] =
TimerValueGet(TIMER0_BASE, TIMER_A);

                    //transpose of data bits
                    i16_Acc_Raw_Data_X = ((pui8_Acc_Read_Data[1] << 8)
                                | pui8_Acc_Read_Data[0]);
```

```
                    i16_Acc_Raw_Data_Y = ((pui8_Acc_Read_Data[3] << 8)
                                | pui8_Acc_Read_Data[2]);
                    i16_Acc_Raw_Data_Z = ((pui8_Acc_Read_Data[5] << 8)
                                | pui8_Acc_Read_Data[4]);

                    //Offsets
                    i16_Acc_Raw_Data_X -= def_ACC_OFFSET_X;
                    i16_Acc_Raw_Data_Y -= def_ACC_OFFSET_Y;
                    i16_Acc_Raw_Data_Z -= def_ACC_OFFSET_Z;

                    //change acceleration data from g to m/s^2 and multiple the
sensor resolution
                    f_Acc_X = (float) i16_Acc_Raw_Data_X * def_ACC_SCALE_FACTOR_X;
// 0.038245935f
                    f_Acc_Y  =  (float)  i16_Acc_Raw_Data_Y  *  def_ACC_SCALE_FACTOR_Y;
                    f_Acc_Z = (float) i16_Acc_Raw_Data_Z * def_ACC_SCALE_FACTOR_Z;

                    CompDCMAccelUpdate(&g_sCompDCMInst, f_Acc_X, f_Acc_Y, f_Acc_Z);

                    /*
                    //Perform moving average

                    //moving average axi X
                    for(ui8_I = 0; ui8_I < def_ACC_NUM_SAMPLES-1; ui8_I++)
                          pi16_Acc_Vector_X[ui8_I] = pi16_Acc_Vector_X[ui8_I+1];
                    pi16_Acc_Vector_X[def_ACC_NUM_SAMPLES-1] = i16_Acc_Raw_Data_X;
                    i32_Acc_Add_X = 0;
                    for(ui8_I = 0; ui8_I < (def_ACC_NUM_SAMPLES); ui8_I++)
                          i32_Acc_Add_X += pi16_Acc_Vector_X[ui8_I];
                    f_Acc_Avg_X = ((float)i32_Acc_Add_X /
(float)def_ACC_NUM_SAMPLES) * def_ACC_SCALE_FACTOR_X;

                    //moving average axi Y
                    for(ui8_I = 0; ui8_I < def_ACC_NUM_SAMPLES-1; ui8_I++)
                          pi16_Acc_Vector_Y[ui8_I] = pi16_Acc_Vector_Y[ui8_I+1];
                    pi16_Acc_Vector_Y[def_ACC_NUM_SAMPLES-1] = i16_Acc_Raw_Data_Y;
                    i32_Acc_Add_Y = 0;
                    for(ui8_I = 0; ui8_I < (def_ACC_NUM_SAMPLES); ui8_I++)
                          i32_Acc_Add_Y += pi16_Acc_Vector_Y[ui8_I];
                    f_Acc_Avg_Y = ((float)i32_Acc_Add_Y /
(float)def_ACC_NUM_SAMPLES);//8* def_ACC_SCALE_FACTOR_Y;

                    //moving average axi Z
                    for(ui8_I = 0; ui8_I < def_ACC_NUM_SAMPLES-1; ui8_I++)
                          pi16_Acc_Vector_Z[ui8_I] = pi16_Acc_Vector_Z[ui8_I+1];
                    pi16_Acc_Vector_Z[def_ACC_NUM_SAMPLES-1] = i16_Acc_Raw_Data_Z;
                    i32_Acc_Add_Z = 0;
                    for(ui8_I = 0; ui8_I < (def_ACC_NUM_SAMPLES); ui8_I++)
                          i32_Acc_Add_Z += pi16_Acc_Vector_Z[ui8_I];
                    f_Acc_Avg_Z = ((float)i32_Acc_Add_Z /
(float)def_ACC_NUM_SAMPLES);//8 * def_ACC_SCALE_FACTOR_Z;

                    //Module calculate
                    f_Acc_Module = sqrtf((f_Acc_Avg_X*f_Acc_Avg_X) +
(f_Acc_Avg_Y*f_Acc_Avg_Y) + (f_Acc_Avg_Z*f_Acc_Avg_Z));


                    //euler angles calculation
```

```
                    f_Ang_X = - asinf(f_Acc_Avg_X / f_Acc_Module) *
def_CONVERT_RADIUS_TO_DEGRESS;
                    f_Ang_Y = - asinf(f_Acc_Avg_Y / f_Acc_Module) *
def_CONVERT_RADIUS_TO_DEGRESS;
                    f_Ang_Z = - asinf(f_Acc_Avg_Z / f_Acc_Module) *
def_CONVERT_RADIUS_TO_DEGRESS;

*/

                    TM_Schedule.tdef_Tasks_Status[PROCESS_ACC] = DONE;

                    TM_Schedule.pui32_Tasks_Finish_Time[PROCESS_ACC]
= TimerValueGet(TIMER0_BASE, TIMER_A);

             }

//*****************************************************************************
//Verify Gyroscope

             if (TM_Schedule.tdef_Tasks_Status[VERIFY_GYRO] == READY) {
                    TM_Schedule.pui32_Tasks_Start_Time[VERIFY_GYRO] =
TimerValueGet(TIMER0_BASE, TIMER_A);

                    //verify the bit ZYXDA-DAta Avaliable in Register 0x09–
STATUS_REG
                    ui8_Data_Ready = I2CReceiveSingleData(GYRO_SLAVE_ADDR, 0x27);
                    if ((ui8_Data_Ready & 0b00001000) > 0) //0b00001000
                                 {
                           TM_Schedule.tdef_Tasks_Status[VERIFY_GYRO] = DONE;
                           TM_Schedule.tdef_Tasks_Status[READ_GYRO] = READY;

                    }

                    TM_Schedule.pui32_Tasks_Finish_Time[VERIFY_GYRO]
= TimerValueGet(TIMER0_BASE, TIMER_A);

             }
//*****************************************************************************
//Read Gyroscope

             if (TM_Schedule.tdef_Tasks_Status[READ_GYRO] == READY) {
                    TM_Schedule.pui32_Tasks_Start_Time[READ_GYRO] =
TimerValueGet(TIMER0_BASE, TIMER_A);

                    //Routine to read gyroscope data by I2C bus
                    I2CReceiveMultipleData(GYRO_SLAVE_ADDR, 0xA8, 6); //00101000b =
0x28 -> 10101000b = 0xA8 make the auto-increment to read multiple data

                    // Routine to atribute buffer of I2C learning
                    for (ui8_I = 0; ui8_I < 6; ui8_I++) {
                           pui8_Gyro_Read_Data[ui8_I] = gpui8_Buffer[ui8_I];

                    }

                    TM_Schedule.tdef_Tasks_Status[READ_GYRO] = DONE;
                    TM_Schedule.tdef_Tasks_Status[PROCESS_GYRO] = READY;

                    TM_Schedule.pui32_Tasks_Finish_Time[READ_GYRO]
= TimerValueGet(TIMER0_BASE, TIMER_A);
```

```
                  //UARTprintf("2");
            }
//*****************************************************************************
//Process Gyroscope

            if (TM_Schedule.tdef_Tasks_Status[PROCESS_GYRO] == READY) {
                  TM_Schedule.pui32_Tasks_Start_Time[PROCESS_GYRO] =
TimerValueGet(TIMER0_BASE, TIMER_A);

                  //transpose of data bits

                  i16_Gyro_Raw_X = ((pui8_Gyro_Read_Data[1] << 8)
                              | pui8_Gyro_Read_Data[0]); // + i16_Gyro_Bias_X;
                  i16_Gyro_Raw_Y = ((pui8_Gyro_Read_Data[3] << 8)
                              | pui8_Gyro_Read_Data[2]);
                  i16_Gyro_Raw_Z = ((pui8_Gyro_Read_Data[5] << 8)
                              | pui8_Gyro_Read_Data[4]);

                  //UARTprintf(";%i;%i;%i\n",i16_Gyro_Raw_X, i16_Gyro_Raw_Y,
i16_Gyro_Raw_Z);

                  //change angular speed from degrees/second to radius/second and
multiple the sensor resolution
                  f_Gyro_X = (float) i16_Gyro_Raw_X * 0.000152716955f; //
0.000152716955 = 8.75mdps/digit * (Pi/180)
                  f_Gyro_Y  = (float)  i16_Gyro_Raw_Y  *  0.000152716955f;
                  f_Gyro_Z = (float) i16_Gyro_Raw_Z * 0.000152716955f;

                  //UARTprintf(";%i", pi16_Gyro_Raw_X[0]);

                  CompDCMGyroUpdate(&g_sCompDCMInst, f_Gyro_X, f_Gyro_Y,
f_Gyro_Z);

                  TM_Schedule.tdef_Tasks_Status[PROCESS_GYRO] = DONE;

                  TM_Schedule.pui32_Tasks_Finish_Time[PROCESS_GYRO] =
TimerValueGet(TIMER0_BASE, TIMER_A);

            }

//*****************************************************************************
//Verify Magnetometer

            if (TM_Schedule.tdef_Tasks_Status[VERIFY_MAG] == READY) {
                  TM_Schedule.pui32_Tasks_Start_Time[VERIFY_MAG] =
TimerValueGet(TIMER0_BASE, TIMER_A);

                  //verify the bit RDY-Ready Bit in Register 0x09—Status Register
                  ui8_Data_Ready = I2CReceiveSingleData(MAG_SLAVE_ADDR, 0x09);
                  //UARTprintf(";%i",ui8_Data_Ready);

                  if ((ui8_Data_Ready & 0b00000001) == 0) //if( (ui8_Data_Ready &
0b00000001) == 0 )
                              {
                        TM_Schedule.tdef_Tasks_Status[VERIFY_MAG] = DONE;
                        TM_Schedule.tdef_Tasks_Status[READ_MAG] = READY;
                  } else {
                        TM_Schedule.tdef_Tasks_Status[VERIFY_MAG] = DONE;
                        TM_Schedule.tdef_Tasks_Status[PROCESS_MAG] = DONE;
```

```
                  }

                  TM_Schedule.pui32_Tasks_Finish_Time[VERIFY_MAG] =
TimerValueGet(TIMER0_BASE, TIMER_A);
            }

//*****************************************************************************
//Read Magnetometer

            if (TM_Schedule.tdef_Tasks_Status[READ_MAG] == READY) {
                  TM_Schedule.pui32_Tasks_Start_Time[READ_MAG] =
TimerValueGet(TIMER0_BASE, TIMER_A);

                  //Routine to read magnetometer data by I2C bus
                  I2CReceiveMultipleData(MAG_SLAVE_ADDR, 0x03, 6);

                  // Routine to atribute buffer of I2C learning
                  for (ui8_I = 0; ui8_I < 6; ui8_I++) {
                        pui8_Mag_Read_Data[ui8_I] = gpui8_Buffer[ui8_I];
                  }

                  TM_Schedule.tdef_Tasks_Status[READ_MAG] = DONE;
                  TM_Schedule.tdef_Tasks_Status[PROCESS_MAG] = READY;

                  TM_Schedule.pui32_Tasks_Finish_Time[READ_MAG] =
TimerValueGet(TIMER0_BASE, TIMER_A);

            }
//*****************************************************************************
//Process Magnetometer

            if (TM_Schedule.tdef_Tasks_Status[PROCESS_MAG] == READY) {
                  TM_Schedule.pui32_Tasks_Start_Time[PROCESS_MAG] =
TimerValueGet(TIMER0_BASE, TIMER_A);

                  // transpose of data bits

                  i16_Mag_Raw_X = ((pui8_Mag_Read_Data[0] << 8)
                              | pui8_Mag_Read_Data[1]);
                  i16_Mag_Raw_Y = ((pui8_Mag_Read_Data[2] << 8)
                              | pui8_Mag_Read_Data[3]);
                  i16_Mag_Raw_Z = ((pui8_Mag_Read_Data[4] << 8)
                              | pui8_Mag_Read_Data[5]);

                  i16_Mag_Raw_X -= def_MAG_OFFSET_X;
                  i16_Mag_Raw_Y -= def_MAG_OFFSET_Y;
                  i16_Mag_Raw_Z -= def_MAG_OFFSET_Z;

                  f_Mag_X = (float) i16_Mag_Raw_X * def_MAG_SCALE_FACTOR_X;
                  f_Mag_Y = (float) i16_Mag_Raw_Y * def_MAG_SCALE_FACTOR_Y;
                  f_Mag_Z = (float) i16_Mag_Raw_Z * def_MAG_SCALE_FACTOR_Z;

                  CompDCMMagnetoUpdate(&g_sCompDCMInst, f_Mag_Y * 0.0001f, -
                              f_Mag_X * 0.0001f, f_Mag_Z * 0.0001f);

                  TM_Schedule.tdef_Tasks_Status[PROCESS_MAG] = DONE;

                  TM_Schedule.pui32_Tasks_Finish_Time[PROCESS_MAG]
= TimerValueGet(TIMER0_BASE, TIMER_A);
```

```
		}

//*****************************************************************************
//Update Complementary Filter DCM and Madgwick Filter

		if (TM_Schedule.tdef_Tasks_Status[FILTER_UPDATE] != DONE
				& TM_Schedule.tdef_Tasks_Status[PROCESS_ACC] == DONE
				& TM_Schedule.tdef_Tasks_Status[PROCESS_GYRO] == DONE
				& TM_Schedule.tdef_Tasks_Status[PROCESS_MAG] == DONE)
		{
			TM_Schedule.tdef_Tasks_Status[FILTER_UPDATE] = READY;
		}

		if (TM_Schedule.tdef_Tasks_Status[FILTER_UPDATE] == READY)
		{
			TM_Schedule.pui32_Tasks_Start_Time[FILTER_UPDATE] =
TimerValueGet(TIMER0_BASE, TIMER_A);

			if (gui8_Fisrt_Reading_Flag == 1)// Perform the sending of
the DCM with the first data set.
			{
				CompDCMStart(&g_sCompDCMInst);
				gui8_Fisrt_Reading_Flag = 0;
			} else
			{
				CompDCMUpdate(&g_sCompDCMInst); // DCM Is already
started. Perform the incremental update.
			}

			CompDCMComputeEulers(&g_sCompDCMInst, &f_Euler_X, &f_Euler_Y,
&f_Euler_Z); //Get Euler angles in radius

			//Convert each angle to degrees
			f_Euler_X = f_Euler_X * (def_CONVERT_RADIUS_TO_DEGRESS);//
def_CONVERT_RADIUS_TO_DEGRESS = 180.0f/M_PI;
			f_Euler_Y  = f_Euler_Y * (def_CONVERT_RADIUS_TO_DEGRESS);
			f_Euler_Z  = f_Euler_Z * (def_CONVERT_RADIUS_TO_DEGRESS);

			//MadgwickAHRSupdate(f_Gyro_X, f_Gyro_Y, f_Gyro_Z, f_Acc_X,
f_Acc_Y, f_Acc_Z, f_Mag_Y, -f_Mag_X, f_Mag_Z);
			MadgwickAHRSupdateIMU(f_Gyro_X, f_Gyro_Y, f_Gyro_Z, f_Acc_X,
f_Acc_Y, f_Acc_Z);


			//******************************** //Proceed
			conversion from quaternion to euler

			/*
			//From Madgwick's paper
			f_Euler_Z2 = atan2f(2.0f * (q1*q2 - q0*q3), 2.0f * (q0*q0 +
q1*q1) - 1);
			f_Euler_X2 = -asinf(2.0f * (q1*q3 + q0*q2));
			f_Euler_Y2 = atan2f(2.0f * (q2*q3 - q0*q1), 2.0f * (q0*q0 +
q3*q3) - 1);


			f_Euler_X2 *= 180.0f / M_PI;
			f_Euler_Y2 *= 180.0f / M_PI;
```

```
                    f_Euler_Z2 *= 180.0f / M_PI;
                    */

                    //From https://github.com/kriswiner/GY-
80/blob/master/GY80BasicAHRS.ino
                    //Almost the same from Tobias Simon in
https://github.com/TobiasSimon/MadgwickTests/blob/master/test.c
                    f_Madgwick_yaw = atan2f(2.0f * (q1 * q2 + q0 * q3), q0 * q0
+ q1 * q1 - q2 * q2 - q3 * q3);
                    f_Madgwick_pitch = -asinf(2.0f * (q1 * q3 - q0 * q2));
                    f_Madgwick_roll = atan2f(2.0f * (q0 * q1 + q2 * q3), q0 * q0 -
q1 * q1 - q2 * q2 + q3 * q3);

                    f_Madgwick_yaw *= def_CONVERT_RADIUS_TO_DEGRESS;
                    //f_Madgwick_yaw -= 13.8; // Declination at location.
                    f_Madgwick_pitch *= def_CONVERT_RADIUS_TO_DEGRESS;
                    f_Madgwick_roll *= def_CONVERT_RADIUS_TO_DEGRESS;

                    TM_Schedule.tdef_Tasks_Status[FILTER_UPDATE] = DONE;

                    TM_Schedule.tdef_Tasks_Status[PID_UPDATE] = READY;

                    TM_Schedule.pui32_Tasks_Finish_Time[FILTER_UPDATE]
= TimerValueGet(TIMER0_BASE, TIMER_A);

             }

//*********************************************************************************
//Update PID

             if (TM_Schedule.tdef_Tasks_Status[PID_UPDATE] == READY)
             {
                    TM_Schedule.pui32_Tasks_Start_Time[PID_UPDATE]
= TimerValueGet(TIMER0_BASE, TIMER_A);

                    if (gui32_Radio_PulseA4 < PWM_MIN)
                           f_Throtle_Porcentage = 0.0f;
                    else if (gui32_Radio_PulseA4 > PWM_MAX)
                           f_Throtle_Porcentage = 100.0f;
                    else
                           f_Throtle_Porcentage = (gui32_Radio_PulseA4 - (PWM_MIN))
* (100.0f / (PWM_MAX - PWM_MIN));

                    ui16_Enable_Control = gui32_Radio_PulseE2;

                    // common for all axis

                    //below check the saturation issue
                    if (f_Throtle_Porcentage <= 50.0f)
                           f_Saturation = f_Throtle_Porcentage;
                    else if (f_Throtle_Porcentage > 50.0f)
                           f_Saturation = 100.0f - f_Throtle_Porcentage;

                    /*
                     Kp = (gui32_Radio_PulseA6 - (PWM_STANDBY)) * (5.0f /
PWM_RANGE);//       5/1024 = 0,004882813f
                     if(Kp < 0)                                                //
remember to change the limits
                     Kp = 0;
```

```
                    if(Kp > 5.0f)
                    Kp = 5.0f;


                    Ki = (gui32_Radio_PulseA7 - (PWM_STANDBY)) * (10.0f /
PWM_RANGE);//      20/1024 = 0.01953125f
                    if(Ki < 0.000f)                                    //remember
to change the limits
                    Ki = 0.000f;
                    if(Ki > 10.0f)
                    Ki = 10.0f;

                    Kd = (gui32_Radio_PulseE3 - (PWM_STANDBY)) * (0.50f /
PWM_RANGE);//      20/1024 = 0.01953125f
                    if(Kd < 0.000f)                                    // remember
to change the limits
                    Kd = 0.000f;
                    if(Kd > 0.50f)
                    Kd = 0.50f;

                    Ki_Calc = (Kp*Kp) / (4*Kd);//if Ki is to small Kd raise to big
value, it closes to infinite

                    */

                   //calc Y axi
                   if (ui16_Enable_Control > PWM_HALF)    //Only sum in case of
motors are armed
                   {
                         /*
                          if(gui8_Second == 5)
                          f_setpoint_Y = 0;
                          if(gui8_Second == 10)
                          f_setpoint_Y = 15;
                          if(gui8_Second == 15)
                          f_setpoint_Y = -15;
                          if(gui8_Second == 20)
                          f_setpoint_Y = 0;
                          */
                          f_setpoint_Y = ((float) gui32_Radio_PulseA3 -
      PWM_HALF) * ANG_RANGE / PWM_RANGE;

                          f_actual_position_Y = f_Madgwick_pitch;

                          //Caculate actual error
                          f_actual_error_Y = f_setpoint_Y - f_actual_position_Y;

                          f_integral_Y += f_actual_error_Y * dt;

                          f_derivative_Y = (f_actual_error_Y - f_pre_error_Y) /
dt; //difference between 2 samples should be 0.5f or smallest. The sensor
fusion do it.

                          f_output_Y = Kp * f_actual_error_Y + Ki * f_integral_Y +
Kd * f_derivative_Y;

                          f_pre_error_Y = f_actual_error_Y;
```

```
                        if (f_output_Y > f_Saturation)//compare to check
saturation on output, the value should be within PWM range
                        {
                              f_output_Y = f_Saturation;
                              f_integral_Y -= f_actual_error_Y * dt;
                        }
                        if (f_output_Y < (-f_Saturation))
                        {
                              f_output_Y = -f_Saturation;
                              f_integral_Y -= f_actual_error_Y * dt;
                        }
                        //else if //keep output PID result

                        f_Output_Left = (((f_Throtle_Porcentage - f_output_Y)
* (PWM_MAX - PWM_MIN)) / 100.0f) + PWM_MIN;
                        f_Output_Right = (((f_Throtle_Porcentage + f_output_Y) *
(PWM_MAX - PWM_MIN)) / 100.0f) + PWM_MIN;

                        TimerMatchSet(WTIMER0_BASE, TIMER_A, (uint32_t)
(SysCtlClockGet() / SAMPLING_FREQUENCY) - (f_Output_Left * 80));
                        TimerMatchSet(WTIMER0_BASE, TIMER_B, (uint32_t)
(SysCtlClockGet() / SAMPLING_FREQUENCY)      - (f_Output_Right * 80));

                        //Keep it for ESC calibration
                        //TimerMatchSet(WTIMER0_BASE, TIMER_BOTH,
(SysCtlClockGet() / SAMPLING_FREQUENCY) - (gui32_Radio_PulseA4 * 80));

                        //calc X axi

                        f_setpoint_X = ((float) gui32_Radio_PulseA2 -
      PWM_HALF) * ANG_RANGE / PWM_RANGE;

                        f_actual_position_X = f_Madgwick_roll;

                        //Caculate actual error
                        f_actual_error_X = f_setpoint_X - f_actual_position_X;

                        f_integral_X += f_actual_error_X * dt;

                        f_derivative_X = (f_actual_error_X - f_pre_error_X) /
dt; //difference between 2 samples should be 0.5f or smallest. The filter
fusion do it.

                        f_output_X = Kp * f_actual_error_X + Ki * f_integral_X
                                    + Kd * f_derivative_X;

                        f_pre_error_X = f_actual_error_X;

                        if (f_output_X > f_Saturation)//compare to check
saturation on output, the value should be within PWM range
                                    {
                              f_output_X = f_Saturation;
                              f_integral_X -= f_actual_error_X * dt;
                        } else if (f_output_X < (-f_Saturation)) {
                              f_output_X = -f_Saturation;
                              f_integral_X -= f_actual_error_X * dt;
                        }
                        //else if //keep output PID result
```

```
                        f_Output_Up = (((f_Throtle_Porcentage - f_output_X)
* (PWM_MAX - PWM_MIN)) / 100.0f) + PWM_MIN;
                        f_Output_Down = (((f_Throtle_Porcentage + f_output_X)*
(PWM_MAX - PWM_MIN)) / 100.0f) + PWM_MIN;

                        //f_Output_Up = (((f_Throtle_Porcentage) * (PWM_MAX -
PWM_MIN)) / 100.0f) + PWM_MIN;
                        //f_Output_Down = (((f_Throtle_Porcentage) * (PWM_MAX -
PWM_MIN)) / 100.0f) + PWM_MIN;

                        TimerMatchSet(WTIMER1_BASE, TIMER_A, (uint32_t)
(SysCtlClockGet() / SAMPLING_FREQUENCY)      - (f_Output_Up * 80));
                        TimerMatchSet(WTIMER1_BASE, TIMER_B, (uint32_t)
(SysCtlClockGet() / SAMPLING_FREQUENCY)      - (f_Output_Down * 80));

                        //Keep it for ESC calibration
                        //TimerMatchSet(WTIMER1_BASE, TIMER_BOTH,
(SysCtlClockGet() / SAMPLING_FREQUENCY) - (gui32_Radio_PulseA4 * 80));

                }
                else//keep motors stopped
                {
                        TimerMatchSet(WTIMER0_BASE, TIMER_BOTH,
                                        (SysCtlClockGet() / SAMPLING_FREQUENCY)
                                                - (PWM_STANDBY * 80));
                        TimerMatchSet(WTIMER1_BASE, TIMER_BOTH,
                                        (SysCtlClockGet() / SAMPLING_FREQUENCY)
                                                - (PWM_STANDBY * 80));

                        f_integral_Y = 0.0f;
                        f_integral_X = 0.0f;
                }

                TM_Schedule.tdef_Tasks_Status[PID_UPDATE] = DONE;

                TM_Schedule.tdef_Tasks_Status[SEND_SERIAL] = READY;

                TM_Schedule.pui32_Tasks_Finish_Time[PID_UPDATE] =
TimerValueGet(TIMER0_BASE, TIMER_A);

        }

//*****************************************************************************
//Send Data through Serial

        if (TM_Schedule.tdef_Tasks_Status[SEND_SERIAL] == READY) {
                TM_Schedule.pui32_Tasks_Start_Time[SEND_SERIAL] =
TimerValueGet(TIMER0_BASE, TIMER_A);

                //UARTprintf(";%i ", GPIOPinRead(GPIO_PORTE_BASE, GPIO_PIN_2)
== GPIO_PIN_2);

                //UARTprintf(";%i\n", gui32_Radio_PulseA2);

                switch (ui8_cycle_count)
                {
                case 0: {

                        UARTprintf(";%i ", gui32_Radio_PulseE2);
```

```
        //UARTprintf(";%i ", (uint32_t) f_Throtle_Porcentage);

        //PrintFloatVariable(f_setpoint_Y, false);

        ui8_cycle_count += 1;
        break;
}
case 1: {
        UARTprintf(";%i ", gui32_Radio_PulseE3);
        //PrintFloatVariable(f_actual_position_Y, false);

        ui8_cycle_count += 1;
        break;
}
case 2: {
        UARTprintf(";%i ", gui32_Radio_PulseA2);
        //PrintFloatVariable(f_actual_error_Y, false);

        ui8_cycle_count += 1;
        break;
}
case 3: {
        UARTprintf(";%i ", gui32_Radio_PulseA3);

        //PrintFloatVariable(f_output_Y, false);

        ui8_cycle_count += 1;
        break;
}
case 4: {
        UARTprintf(";%i ", gui32_Radio_PulseA4);
        //UARTprintf(";%i", gui8_Second);

        ui8_cycle_count += 1;
        break;
}
case 5: {
        UARTprintf(";%i ", gui32_Radio_PulseA5);
        //PrintFloatVariable(f_setpoint_X, false);

        ui8_cycle_count += 1;
        break;
}
case 6: {
        UARTprintf(";%i ", gui32_Radio_PulseA6);
        //PrintFloatVariable(f_actual_position_X, false);

        ui8_cycle_count += 1;
        break;
}
case 7: {
        UARTprintf(";%i\n ", gui32_Radio_PulseA7);
        //PrintFloatVariable(f_actual_error_X, false);

        ui8_cycle_count += 1;
        break;
}
case 8: {
```

```
                        //PrintFloatVariable(f_output_X, true);

                        ui8_cycle_count += 1;
                        break;
                }
                case 9: {

                        ui8_cycle_count = 0;
                        break;
                }

                }// end switch

                TM_Schedule.tdef_Tasks_Status[SEND_SERIAL] = DONE;
                TM_Schedule.pui32_Tasks_Finish_Time[SEND_SERIAL] =
TimerValueGet(TIMER0_BASE, TIMER_A);

            }//end task

//*****************************************************************************

        }//end for(ever)

}//end main

//*****************************************************************************

void InitTimer0(void) {
        // Config and enable Timer0
        SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);

        TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);

        uint32_t ui32Period;
        ui32Period = (uint32_t) (SysCtlClockGet() / SAMPLING_FREQUENCY);
        TimerLoadSet(TIMER0_BASE, TIMER_A, ui32Period - 1);

        TimerControlStall(TIMER0_BASE, TIMER_A, true);
        //IntPrioritySet(INT_TIMER0A, 10); //Priority range is 0 for highest and
255 for lowest
        TimerIntRegister(TIMER0_BASE, TIMER_A, Timer0IntHandler);
        IntEnable(INT_TIMER0A);
        TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
        IntMasterEnable();

        TimerEnable(TIMER0_BASE, TIMER_A);
        UARTprintf("Timer0 working\n");
}

//*****************************************************************************

void Timer0IntHandler(void) {
        // Clear the timer interrupt
        TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

        //TimerSynchronize(TIMER0_BASE, WTIMER_0A_SYNC | WTIMER_0B_SYNC );//
| TIMER_2A_SYNC | TIMER_2B_SYNC

        gui16_AuxTick += 1;
```

```
	if (gui16_AuxTick == (uint16_t) SAMPLING_FREQUENCY)
	{
		gui8_Second += 1;
		gui16_AuxTick = 0;
		if (gui8_Second > 200)
			gui8_Second = 201;

		// Read the current state of the GPIO pin and write back the opposite
state
		if (GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_2)) {
			GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);
		} else {
			GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 4);// if comented
keep off the led
		}
	}

	if ((TM_Schedule.tdef_Tasks_Status[SEND_SERIAL] == DONE)
| (gui8_First_Timeout_Flag == 1))
	{
		gui8_First_Timeout_Flag = 0;
		TM_Schedule.tdef_Tasks_Status[INITIAL] = READY;
	}
	else
	{

		TimerDisable(WTIMER0_BASE, TIMER_BOTH);
		TimerDisable(WTIMER1_BASE, TIMER_BOTH);

		UARTprintf("Timer0 interrupt...TIMEOUT\n");

		while (1) {
		//TIMEOUT
		}

	}

}

//*****************************************************************************

void InitWTimer0PWM(void)
{
	SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
	SysCtlDelay(5);

	GPIOPinConfigure(GPIO_PC4_WT0CCP0);
	GPIOPinTypeTimer(GPIO_PORTC_BASE, GPIO_PIN_4);

	GPIOPinConfigure(GPIO_PC5_WT0CCP1);
	GPIOPinTypeTimer(GPIO_PORTC_BASE, GPIO_PIN_5);

	// Config and enable Timer0
	SysCtlPeripheralEnable(SYSCTL_PERIPH_WTIMER0);
	SysCtlDelay(5);

	TimerConfigure(WTIMER0_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_A_PWM |
TIMER_CFG_B_PWM);
```

```
	TimerLoadSet(WTIMER0_BASE, TIMER_BOTH, SysCtlClockGet() /
SAMPLING_FREQUENCY);

	TimerMatchSet(WTIMER0_BASE, TIMER_BOTH, (SysCtlClockGet() /
SAMPLING_FREQUENCY) - (PWM_STANDBY * 80));//it is important

	TimerUpdateMode(WTIMER0_BASE, TIMER_BOTH,
	TIMER_UP_LOAD_TIMEOUT); TimerControlStall(WTIMER0_BASE,
	TIMER_BOTH, true); //TimerDisable(WTIMER0_BASE, TIMER_BOTH);
	TimerEnable(WTIMER0_BASE, TIMER_BOTH);

}

void InitWTimer1PWM(void)
{

	SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
	SysCtlDelay(5);

	GPIOPinConfigure(GPIO_PC6_WT1CCP0);
	GPIOPinTypeTimer(GPIO_PORTC_BASE, GPIO_PIN_6);

	GPIOPinConfigure(GPIO_PC7_WT1CCP1);
	GPIOPinTypeTimer(GPIO_PORTC_BASE, GPIO_PIN_7);

	// Config and enable Timer0
	SysCtlPeripheralEnable(SYSCTL_PERIPH_WTIMER1);
	SysCtlDelay(5);

	TimerConfigure(WTIMER1_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_A_PWM
| TIMER_CFG_B_PWM);

	TimerLoadSet(WTIMER1_BASE, TIMER_BOTH, SysCtlClockGet() /
SAMPLING_FREQUENCY);

	TimerMatchSet(WTIMER1_BASE, TIMER_BOTH, (SysCtlClockGet() /
SAMPLING_FREQUENCY) - (PWM_STANDBY * 80));//it is important

	TimerUpdateMode(WTIMER1_BASE, TIMER_BOTH, TIMER_UP_LOAD_TIMEOUT);
	TimerControlStall(WTIMER1_BASE, TIMER_BOTH, true);

	TimerEnable(WTIMER1_BASE, TIMER_BOTH);

}

//*****************************************************************************

void InitRadioInput() {

	SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
	SysCtlDelay(5);
	GPIOPinTypeGPIOInput(GPIO_PORTE_BASE, GPIO_PIN_2 | GPIO_PIN_3);
	GPIOIntTypeSet(GPIO_PORTE_BASE, GPIO_PIN_2 | GPIO_PIN_3, GPIO_BOTH_EDGES);
	GPIOIntRegister(GPIO_PORTE_BASE, PortEIntHandler);
	//IntPrioritySet(INT_GPIOE, 14); //Priority range is 0 for highest and 255
for lowest
	GPIOIntEnable(GPIO_PORTE_BASE, GPIO_PIN_2 | GPIO_PIN_3);

	SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
```

```
	SysCtlDelay(5);
	GPIOPinTypeGPIOInput(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_4
| GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7);
	GPIOIntTypeSet(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_4 |
GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7, GPIO_BOTH_EDGES);
	GPIOIntRegister(GPIO_PORTA_BASE, PortAIntHandler);
	//IntPrioritySet(INT_GPIOA, 9); //Priority range is 0 for highest and 255
for lowest
	GPIOIntEnable(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_4 |
GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7);

	//Set the timers to be periodic.
	SysCtlPeripheralEnable(SYSCTL_PERIPH_WTIMER2);
	SysCtlDelay(5);
	TimerConfigure(WTIMER2_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_A_PERIODIC
| TIMER_CFG_B_PERIODIC);
	TimerEnable(WTIMER2_BASE, TIMER_BOTH);

	SysCtlPeripheralEnable(SYSCTL_PERIPH_WTIMER3);
	SysCtlDelay(5);
	TimerConfigure(WTIMER3_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_A_PERIODIC |
TIMER_CFG_B_PERIODIC);
	TimerEnable(WTIMER3_BASE, TIMER_BOTH);

	SysCtlPeripheralEnable(SYSCTL_PERIPH_WTIMER4);
	SysCtlDelay(5);
	TimerConfigure(WTIMER4_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_A_PERIODIC |
TIMER_CFG_B_PERIODIC);
	TimerEnable(WTIMER4_BASE, TIMER_BOTH);

	SysCtlPeripheralEnable(SYSCTL_PERIPH_WTIMER5);
	SysCtlDelay(5);
	TimerConfigure(WTIMER5_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_A_PERIODIC
| TIMER_CFG_B_PERIODIC);
	TimerEnable(WTIMER5_BASE, TIMER_BOTH);
}

//*****************************************************************************

void PortEIntHandler() {
	//Get interrupt flag
	ui32_IntStatus = GPIOIntStatus(GPIO_PORTE_BASE, true);

	if ((ui32_IntStatus & GPIO_PIN_2) == GPIO_PIN_2) {
		GPIOIntClear(GPIO_PORTE_BASE, GPIO_PIN_2);   //This way avoid
throbles
		if (GPIOPinRead (GPIO_PORTE_BASE, GPIO_PIN_2) == GPIO_PIN_2)       //If
it's a rising edge then set the timer to any value to calculate after
		{
			TimerLoadSet(WTIMER2_BASE, TIMER_A, 1000000);
			TimerEnable(WTIMER2_BASE, TIMER_A);
		}

		else if (GPIOPinRead(GPIO_PORTE_BASE, GPIO_PIN_2) == 0) //If it's
a falling edge that was detected, then get the value of the counter
		{
			gui32_Radio_PulseE2 = 1000000 - TimerValueGet(WTIMER2_BASE,
TIMER_A); //record value
			TimerDisable(WTIMER2_BASE, TIMER_A);
```

```
                    //if(((gui32_Radio_PulseE2 / 80) > 100) & ((gui32_Radio_PulseE2
/ 80) < 10000))
                    gui32_Radio_PulseE2 = (uint32_t) (gui32_Radio_PulseE2 / 80); //
80M/1M = 80, so get the time in micro-seconds
                    //UARTprintf("e2=%2d\n" , gui32_Radio_PulseE2);//Prints out the
time measured. Range: 988 to 2012, middle at 15000
              }
       }

       else if ((ui32_IntStatus & GPIO_PIN_3) == GPIO_PIN_3) {
              GPIOIntClear(GPIO_PORTE_BASE, GPIO_PIN_3);
              if (GPIOPinRead(GPIO_PORTE_BASE, GPIO_PIN_3) == GPIO_PIN_3)
              {
                    TimerLoadSet(WTIMER2_BASE, TIMER_B, 1000000);
                    TimerEnable(WTIMER2_BASE, TIMER_B);
              }
              else if (GPIOPinRead(GPIO_PORTE_BASE, GPIO_PIN_3) == 0)
              {
                    gui32_Radio_PulseE3 = 1000000 - TimerValueGet(WTIMER2_BASE,
TIMER_B); //record value
                    TimerDisable(WTIMER2_BASE, TIMER_B);
                    gui32_Radio_PulseE3 = (uint32_t) (gui32_Radio_PulseE3 / 80); //
80M/1M = 80, so get the time in micro-seconds
                    //UARTprintf("e3=%2d\n" , gui32_Radio_PulseE3);//Prints out the
time measured. Range: 988 to 2012
              }
       }
}

void PortAIntHandler() {
       //Clear interrupt flag
       ui32_IntStatus = GPIOIntStatus(GPIO_PORTA_BASE, true);

       if ((ui32_IntStatus & GPIO_PIN_2) == GPIO_PIN_2) {
              GPIOIntClear(GPIO_PORTA_BASE, GPIO_PIN_2);   //This way avoid
throbles
              if (GPIOPinRead(GPIO_PORTA_BASE, GPIO_PIN_2) ==                 //If
GPIO_PIN_2) it's a rising edge then set he timer to any value to
calculate after {
                    TimerLoadSet(WTIMER3_BASE, TIMER_A, 1000000);
                    TimerEnable(WTIMER3_BASE, TIMER_A);
              }

              else if (GPIOPinRead(GPIO_PORTA_BASE, GPIO_PIN_2) == 0) //If it's
a falling edge that was detected, then get the value of the counter
              {
                    gui32_Radio_PulseA2 = 1000000 - TimerValueGet(WTIMER3_BASE,
TIMER_A); //record value
                    TimerDisable(WTIMER3_BASE, TIMER_A);
                    gui32_Radio_PulseA2 = (uint32_t) (gui32_Radio_PulseA2 / 80); //
80M/1M = 80, so get the time in micro-seconds
                    //UARTprintf("a2=%2d\n" , gui32_Radio_PulseA2);//Prints out the
time measured. Range: 988 to 2012, middle at 1500
              }
       }

       else if ((ui32_IntStatus & GPIO_PIN_3) == GPIO_PIN_3) {
              GPIOIntClear(GPIO_PORTA_BASE, GPIO_PIN_3);
              if (GPIOPinRead(GPIO_PORTA_BASE, GPIO_PIN_3) == GPIO_PIN_3)
```

```
            {
                  TimerLoadSet(WTIMER3_BASE, TIMER_B, 1000000);
                  TimerEnable(WTIMER3_BASE, TIMER_B);
            }
            else if (GPIOPinRead(GPIO_PORTA_BASE, GPIO_PIN_3) == 0) {
                  gui32_Radio_PulseA3 = 1000000
                              - TimerValueGet(WTIMER3_BASE, TIMER_B); //record
value
                  TimerDisable(WTIMER3_BASE, TIMER_B);
                  gui32_Radio_PulseA3 = (uint32_t) (gui32_Radio_PulseA3 / 80); //
80M/1M = 80, so get the time in micro-seconds
                  //UARTprintf("a3=%2d\n" , gui32_Radio_PulseA3);//Prints out the
time measured. Range: 988 to 2012
            }
      }

       else if ((ui32_IntStatus & GPIO_PIN_4) == GPIO_PIN_4)
           { GPIOIntClear(GPIO_PORTA_BASE, GPIO_PIN_4);        //This way avoid
throble
            if (GPIOPinRead(GPIO_PORTA_BASE, GPIO_PIN_4) ==                 //If
GPIO_PIN_4) it's a rising edge then set he timer to any value to
calculate after {
                  TimerLoadSet(WTIMER4_BASE, TIMER_A, 1000000);
                  TimerEnable(WTIMER4_BASE, TIMER_A);
            }

            else if (GPIOPinRead(GPIO_PORTA_BASE, GPIO_PIN_4) == 0) //If it's
a falling edge that was detected, then get the value of the counter
            {
                  gui32_Radio_PulseA4 = 1000000 - TimerValueGet(WTIMER4_BASE,
TIMER_A); //record value
                  TimerDisable(WTIMER4_BASE, TIMER_A);
                  gui32_Radio_PulseA4 = (uint32_t) (gui32_Radio_PulseA4 / 80); //
80M/1M = 80, so get the time in micro-seconds
                  //UARTprintf("a4=%2d\n" , gui32_Radio_PulseA4);//Prints out the
time measured.
            }
      }

      else if ((ui32_IntStatus & GPIO_PIN_5) == GPIO_PIN_5) {
            GPIOIntClear(GPIO_PORTA_BASE, GPIO_PIN_5);
            if (GPIOPinRead(GPIO_PORTA_BASE, GPIO_PIN_5) == GPIO_PIN_5)
            {
                  TimerLoadSet(WTIMER4_BASE, TIMER_B, 1000000);
                  TimerEnable(WTIMER4_BASE, TIMER_B);
            }
            else if (GPIOPinRead(GPIO_PORTA_BASE, GPIO_PIN_5) == 0)
            {
                  gui32_Radio_PulseA5 = 1000000 - TimerValueGet(WTIMER4_BASE,
TIMER_B); //record value
                  TimerDisable(WTIMER4_BASE, TIMER_B);
                  gui32_Radio_PulseA5 = (uint32_t) (gui32_Radio_PulseA5 / 80); //
80M/1M = 80, so get the time in micro-seconds
                  //UARTprintf("a5=%2d\n" , gui32_Radio_PulseA5);//Prints out the
time measured.
            }
      }
      else if ((ui32_IntStatus & GPIO_PIN_6) == GPIO_PIN_6)
      {
```

```
            GPIOIntClear(GPIO_PORTA_BASE, GPIO_PIN_6);      //This way avoid
throbles
            if (GPIOPinRead(GPIO_PORTA_BASE, GPIO_PIN_6) ==                  //If
GPIO_PIN_6) it's a rising edge then set he timer to any value to
calculate after {
                  TimerLoadSet(WTIMER5_BASE, TIMER_A, 1000000);
                  TimerEnable(WTIMER5_BASE, TIMER_A);
            }

            else if (GPIOPinRead(GPIO_PORTA_BASE, GPIO_PIN_6) == 0) //If it's
a falling edge that was detected, then get the value of the counter
                        {
                  gui32_Radio_PulseA6 = 1000000
                              - TimerValueGet(WTIMER5_BASE, TIMER_A); //record
value
                  TimerDisable(WTIMER5_BASE, TIMER_A);
                  gui32_Radio_PulseA6 = (uint32_t) (gui32_Radio_PulseA6 / 80); //
80M/1M = 80, so get the time in micro-seconds
                  //UARTprintf("a6=%2d\n" , gui32_Radio_PulseA6);//Prints out the
time measured.
            }
      }

      else if ((ui32_IntStatus & GPIO_PIN_7) == GPIO_PIN_7) {
            GPIOIntClear(GPIO_PORTA_BASE, GPIO_PIN_7);
            if (GPIOPinRead(GPIO_PORTA_BASE, GPIO_PIN_7) == GPIO_PIN_7) {
                  TimerLoadSet(WTIMER5_BASE, TIMER_B, 1000000);
                  TimerEnable(WTIMER5_BASE, TIMER_B);
            } else if (GPIOPinRead(GPIO_PORTA_BASE, GPIO_PIN_7) == 0) {
                  gui32_Radio_PulseA7 = 1000000
                              - TimerValueGet(WTIMER5_BASE, TIMER_B); //record
value
                  TimerDisable(WTIMER5_BASE, TIMER_B);
                  gui32_Radio_PulseA7 = (uint32_t) (gui32_Radio_PulseA7 / 80); //
80M/1M = 80, so get the time in micro-seconds
                  //UARTprintf("a7=%2d\n" , gui32_Radio_PulseA7);//Prints out the
time measured.
            }
      }
}

//*****************************************************************************

void InitLedSWs(void) {

      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
      SysCtlDelay(5);
      // Config Ports function
      GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2
| GPIO_PIN_3); // RGB led

      HWREG(GPIO_PORTF_BASE + GPIO_O_LOCK) = GPIO_LOCK_KEY;
      HWREG(GPIO_PORTF_BASE + GPIO_O_CR) |= 0x01;
      HWREG(GPIO_PORTF_BASE + GPIO_O_LOCK) = 0;
      GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, GPIO_PIN_4 | GPIO_PIN_0); // Sw1 &
Sw2 buttons with internal pull down of 2mA
      GPIOPadConfigSet(GPIO_PORTF_BASE, GPIO_PIN_4 | GPIO_PIN_0,
  GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);
      UARTprintf("Led and SW working\n");
```

```
}

//*****************************************************************************

void InitUART0(void) {
	//Enable UART0 so that we can configure the clock.
	SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
	SysCtlDelay(5);

	//Enable GPIO port A which is used for UART0 pins.
	SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
	SysCtlDelay(5);
	//Configure the pin muxing for UART0 functions on port A0 and A1.
	GPIOPinConfigure(GPIO_PA0_U0RX);
	GPIOPinConfigure(GPIO_PA1_U0TX);

	// Use the internal 16MHz oscillator as the UART clock source.
	UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC);
	// Select the alternate (UART) function for these pins.
	GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

	// Initialize the UART for console I/O.
	UARTStdioConfig(0, 115200, 16000000);
	UARTprintf("UART working... \n ");
}

//*****************************************************************************

void InitADXL345() {

	//Setting up the accelerometer ADXL345, slave_addr = 0x53

	I2CSendSingleData(ACCEL_SLAVE_ADDR, 0x2C, 0x0C);// Data rate and power mode
control: reg_addr = 0x2C, value = 0x0C to 400 Hz of Data Rate
	I2CSendSingleData(ACCEL_SLAVE_ADDR, 0x2D, 0x08);//Power-saving features
control: reg_addr = 0x2D value = 0x08 to measurement modo
	I2CSendSingleData(ACCEL_SLAVE_ADDR, 0x2E, 0x00);//Interrupt control:
reg_addr = 0x2E, value = 0x00 to all disable
	I2CSendSingleData(ACCEL_SLAVE_ADDR, 0x31, 0x0B);//Data format: reg_addr =
0x31 value = 0x08 to full resolution and 2g of range

}

void InitL3G4200D() {

	//Setting up the gyroscope L3G4200D, slave_addr = 0x69

	//with LPF1: 78 Hz and LPF2: 20 Hz

	I2CSendSingleData(GYRO_SLAVE_ADDR, 0x21, 0x05); //High-pass filter:
reg_addr = 0x21 value = 00000101b = 0x05 to normal mode and cut-off 1hz
I2CSendSingleData(GYRO_SLAVE_ADDR, 0x23, 0x80); //Data and scale: reg_addr =
0x23, value = 10000000b = 0x80 to Block Data Update and full resolution
250dps. Self test: 0x82 or 0x86

	I2CSendSingleData(GYRO_SLAVE_ADDR, 0x2E, 0x00);     //FIFO mode selection:
reg_addr = 0x2E value =  00000000b = 0x00 to bypass mode
```

```
	I2CSendSingleData(GYRO_SLAVE_ADDR, 0x24, 0x02); //Filter and
interuption: reg_addr = 0x24 value = 00000010b = 0x02 to enable LPF2 and HPF
I2CSendSingleData(GYRO_SLAVE_ADDR, 0x20, 0x8F); // Data rate and low-
pass filter 2: reg_addr = 0x20, value = 10001111b =0x8F to 400 Hz of Data Rate and
low pass 2 cut-off at 20hz

}

void InitHMC588L() {

	//Setting up the compass HMC588L, slave_addr = 0x1E

	I2CSendSingleData(MAG_SLAVE_ADDR, 0x00, 0x58);////00011000b changed
to 01011000b <- to average 4 samples
	I2CSendSingleData(MAG_SLAVE_ADDR, 0x01, 0x20);//defalt value -> 1.3 Ga ->
Gain:1090 -> Resolution: 0.92
	I2CSendSingleData(MAG_SLAVE_ADDR, 0x02, 0x00);//continuous-measurement mode

}

//*****************************************************************************

//Adapted from eewiki.net
//Slightly modified version of TI's example code

//initialize I2C module 0
void InitI2C0(void) {
	//enable I2C module 0
	SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);

	//reset module
	SysCtlPeripheralReset(SYSCTL_PERIPH_I2C0);

	//enable GPIO peripheral that contains I2C 0
	SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

	// Configure the pin muxing for I2C0 functions on port B2 and
	B3. GPIOPinConfigure(GPIO_PB2_I2C0SCL);
	GPIOPinConfigure(GPIO_PB3_I2C0SDA);

	// Select the I2C function for these pins.
	GPIOPinTypeI2CSCL(GPIO_PORTB_BASE, GPIO_PIN_2);
	GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_3);

	// Enable and initialize the I2C0 master module.  Use the system clock for
	// the I2C0 module.  The last parameter sets the I2C data transfer rate.
	// If false the data rate is set to 100kbps and if true the data rate will
	// be set to 400kbps.
	I2CMasterInitExpClk(I2C0_BASE, SysCtlClockGet(), true);

	//clear I2C FIFOs
	HWREG(I2C0_BASE + I2C_O_FIFOCTL) = 80008000;
}

//sends an I2C command to the specified slave
void I2CSendSingleData(uint8_t ui8SlaveAddr, uint8_t ui8RegAddr, uint8_t ui8Data)
{
	// Tell the master module what address it will place on the bus when
	// communicating with the slave.
```

```
        I2CMasterSlaveAddrSet(I2C0_BASE, ui8SlaveAddr, false);

        I2CMasterDataPut(I2C0_BASE, ui8RegAddr);

        //Initiate send of data from the MCU
        I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_START);

        // Wait until MCU is done transferring.
        while (I2CMasterBusy(I2C0_BASE));

        //Put last and second piece of data into I2C
        FIFO I2CMasterDataPut(I2C0_BASE, ui8Data);

        //send next data that was just placed into FIFO
        I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_FINISH);

        // Wait until MCU is done transferring.
        while (I2CMasterBusy(I2C0_BASE));

        //Send to UART the confirmation
        UARTprintf("In address %02x sent data %02x \n", ui8RegAddr, ui8Data);

}

//read specified register on slave device
uint32_t I2CReceiveSingleData(uint8_t ui8SlaveAddr, uint8_t ui8RegAddr)
        { //specify that we are writing (a register address) to the
        //slave device
        I2CMasterSlaveAddrSet(I2C0_BASE, ui8SlaveAddr, false);

        //specify register to be read
        I2CMasterDataPut(I2C0_BASE, ui8RegAddr);

        //send control byte and register address byte to slave device
        I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_START);

        //wait for MCU to finish transaction
        while (I2CMasterBusy(I2C0_BASE));

        //specify that we are going to read from slave device
        I2CMasterSlaveAddrSet(I2C0_BASE, ui8SlaveAddr, true);

        //send control byte and read from the register we specified
        I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);

        //wait for MCU to finish transaction
        while (I2CMasterBusy(I2C0_BASE));

        //return data pulled from the specified
        register return I2CMasterDataGet(I2C0_BASE);

}

//read multiple data from slave
//If the slave device has not auto-incrementation you shoud search in
the datasheet what do
void I2CReceiveMultipleData(uint8_t ui8SlaveAddr, uint8_t ui8FirstReg, uint8_t
ui8NumReg) {
        //specify that we are writing (a register address) to the slave device
```

```
	I2CMasterSlaveAddrSet(I2C0_BASE, ui8SlaveAddr, false);

	//specify register to be read
	I2CMasterDataPut(I2C0_BASE, ui8FirstReg);

	//send control byte and register address byte to slave device
	I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_START);

	//wait for MCU to finish transaction
	while (I2CMasterBusy(I2C0_BASE));

	//specify that we are going to read from slave device
	I2CMasterSlaveAddrSet(I2C0_BASE, ui8SlaveAddr, true);

	if (ui8NumReg == 1) {
		//send control byte and read from the register we specified
		I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);

		//wait for MCU to finish transaction
		while (I2CMasterBusy(I2C0_BASE));

		//return data pulled from the specified register
		gpui8_Buffer[0] = I2CMasterDataGet(I2C0_BASE);
	}
	else
	{
		//send control byte and read from the register we specified
		I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_RECEIVE_START);

		//wait for MCU to finish transaction
		while (I2CMasterBusy(I2C0_BASE));

		//return data pulled from the specified register
		gpui8_Buffer[0] = I2CMasterDataGet(I2C0_BASE);

		uint8_t ui8I;
		for (ui8I = 1; ui8I < ui8NumReg; ui8I++) {

			//send control byte and read from the register we specified
			I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_RECEIVE_CONT);

			//wait for MCU to finish transaction
			while (I2CMasterBusy(I2C0_BASE));

			//return data pulled from the specified register
			gpui8_Buffer[ui8I] = I2CMasterDataGet(I2C0_BASE);
		}

		//send control byte and read from the register we specified
		I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_RECEIVE_FINISH);

		//wait for MCU to finish transaction
		while (I2CMasterBusy(I2C0_BASE));

		//return data pulled from the specified register
		I2CMasterDataGet(I2C0_BASE); // i call this function only to take
off last data from the FIFO
	}
```

```
}

//============================================================================
// AHRS algorithm update

void MadgwickAHRSupdate(float gx, float gy, float gz, float ax, float ay, float
az, float mx, float my, float mz)
{
	float recipNorm;
	float s0, s1, s2, s3;
	float qDot1, qDot2, qDot3, qDot4;
	float hx, hy;
	float _2q0mx, _2q0my, _2q0mz, _2q1mx, _2bx, _2bz, _4bx, _4bz, _2q0, _2q1,
		_2q2, _2q3, _2q0q2, _2q2q3, q0q0, q0q1, q0q2, q0q3, q1q1, q1q2,
		q1q3, q2q2, q2q3, q3q3;

	// Use IMU algorithm if magnetometer measurement invalid (avoids NaN in
magnetometer normalisation)
	if ((mx == 0.0f) && (my == 0.0f) && (mz == 0.0f)) {
		//MadgwickAHRSupdateIMU(gx, gy, gz, ax, ay, az);
		UARTprintf("some problem with magnetometer");
		return;
	}

	// Rate of change of quaternion from gyroscope
	qDot1 = 0.5f * (-q1 * gx - q2 * gy - q3 * gz);
	qDot2 = 0.5f * (q0 * gx + q2 * gz - q3 * gy);
	qDot3 = 0.5f * (q0 * gy - q1 * gz + q3 * gx);
	qDot4 = 0.5f * (q0 * gz + q1 * gy - q2 * gx);

	// Compute feedback only if accelerometer measurement valid (avoids NaN in
accelerometer normalisation)
	if (!((ax == 0.0f) && (ay == 0.0f) && (az == 0.0f))) {

		// Normalise accelerometer measurement
		recipNorm = invSqrt(ax * ax + ay * ay + az * az);
		ax *= recipNorm;
		ay *= recipNorm;
		az *= recipNorm;

		// Normalise magnetometer measurement
		recipNorm = invSqrt(mx * mx + my * my + mz * mz);
		mx *= recipNorm;
		my *= recipNorm;
		mz *= recipNorm;

		// Auxiliary variables to avoid repeated arithmetic
		_2q0mx = 2.0f * q0 * mx;
		_2q0my = 2.0f * q0 * my;
		_2q0mz = 2.0f * q0 * mz;
		_2q1mx = 2.0f * q1 * mx;
		_2q0 = 2.0f * q0;
		_2q1 = 2.0f * q1;
		_2q2 = 2.0f * q2;
		_2q3 = 2.0f * q3;
		_2q0q2 = 2.0f * q0 * q2;
		_2q2q3 = 2.0f * q2 * q3;
		q0q0 = q0 * q0;
		q0q1 = q0 * q1;
```

```
q0q2 = q0 * q2;
q0q3 = q0 * q3;
q1q1 = q1 * q1;
q1q2 = q1 * q2;
q1q3 = q1 * q3;
q2q2 = q2 * q2;
q2q3 = q2 * q3;
q3q3 = q3 * q3;

// Reference direction of Earth's magnetic field
hx = mx * q0q0 - _2q0my * q3 + _2q0mz * q2
       + mx * q1q1 + _2q1 * my * q2
       + _2q1 * mz * q3 - mx * q2q2 - mx * q3q3;
hy = _2q0mx * q3 + my * q0q0 - _2q0mz * q1
             + _2q1mx * q2 - my * q1q1
             + my * q2q2 + _2q2 * mz * q3 - my *
q3q3; _2bx = sqrt(hx * hx + hy * hy);
_2bz = -_2q0mx * q2 + _2q0my * q1 + mz * q0q0
             + _2q1mx * q3 - mz * q1q1
             + _2q2 * my * q3 - mz * q2q2 + mz *
q3q3; _4bx = 2.0f * _2bx;
_4bz = 2.0f * _2bz;

// Gradient decent algorithm corrective step
s0 = -_2q2 * (2.0f * q1q3 - _2q0q2 - ax)
+ _2q1 * (2.0f * q0q1 + _2q2q3 - ay)
- _2bz * q2  * (_2bx * (0.5f - q2q2 - q3q3) + _2bz * (q1q3 - q0q2)
- mx)
+ (-_2bx * q3 + _2bz * q1)* (_2bx * (q1q2 - q0q3)
+ _2bz * (q0q1 + q2q3) - my)
+ _2bx * q2 * (_2bx * (q0q2 + q1q3) + _2bz * (0.5f - q1q1 - q2q2)
- mz);

s1 = _2q3 * (2.0f * q1q3 - _2q0q2 - ax)
+ _2q0 * (2.0f * q0q1 + _2q2q3 - ay)
- 4.0f * q1 * (1 - 2.0f * q1q1 - 2.0f * q2q2 - az)
+ _2bz * q3 * (_2bx * (0.5f - q2q2 - q3q3) + _2bz * (q1q3 - q0q2)
- mx)
+ (_2bx * q2 + _2bz * q0) * (_2bx * (q1q2 - q0q3)
+ _2bz * (q0q1 + q2q3)
- my)
+ (_2bx * q3 - _4bz * q1) * (_2bx * (q0q2 + q1q3)
+ _2bz * (0.5f - q1q1 - q2q2)
- mz);

s2 = -_2q0 * (2.0f * q1q3 - _2q0q2 - ax)
+ _2q3 * (2.0f * q0q1 + _2q2q3 - ay)
- 4.0f * q2 * (1 - 2.0f * q1q1 - 2.0f * q2q2 - az)
+ (-_4bx * q2 - _2bz * q0) * (_2bx * (0.5f - q2q2 - q3q3)
+ _2bz * (q1q3 - q0q2)
- mx)
+ (_2bx * q1 + _2bz * q3) * (_2bx * (q1q2 - q0q3)
+ _2bz * (q0q1 + q2q3)
- my)
+ (_2bx * q0 - _4bz * q2) * (_2bx * (q0q2 + q1q3)
+ _2bz * (0.5f - q1q1 - q2q2)
- mz);

s3 = _2q1 * (2.0f * q1q3 - _2q0q2 - ax)
```

```
            + _2q2 * (2.0f * q0q1 + _2q2q3 - ay)
            + (-_4bx * q3 + _2bz * q1) * (_2bx * (0.5f - q2q2 - q3q3)
            + _2bz * (q1q3 - q0q2)
            - mx)
            + (-_2bx * q0 + _2bz * q2) * (_2bx * (q1q2 - q0q3)
            + _2bz * (q0q1 + q2q3) - my)
            + _2bx * q1 * (_2bx * (q0q2 + q1q3)
            + _2bz * (0.5f - q1q1 - q2q2)
            - mz);

            recipNorm = invSqrt(s0 * s0 + s1 * s1 + s2 * s2 + s3 * s3); //
normalise step magnitude
            s0 *= recipNorm;
            s1 *= recipNorm;
            s2 *= recipNorm;
            s3 *= recipNorm;

            // Apply feedback step
            qDot1 -= beta * s0;
            qDot2 -= beta * s1;
            qDot3 -= beta * s2;
            qDot4 -= beta * s3;
      }

      // Integrate rate of change of quaternion to yield quaternion
      q0 += qDot1 * (1.0f / SAMPLING_FREQUENCY);
      q1 += qDot2 * (1.0f / SAMPLING_FREQUENCY);
      q2 += qDot3 * (1.0f / SAMPLING_FREQUENCY);
      q3 += qDot4 * (1.0f / SAMPLING_FREQUENCY);

      // Normalise quaternion
      recipNorm = invSqrt(q0 * q0 + q1 * q1 + q2 * q2 + q3 * q3);
      q0 *= recipNorm;
      q1 *= recipNorm;
      q2 *= recipNorm;
      q3 *= recipNorm;
}

//****************************************************************************
// IMU algorithm update

void MadgwickAHRSupdateIMU(float gx, float gy, float gz, float ax, float
            ay, float az) {
      float recipNorm;
      float s0, s1, s2, s3;
      float qDot1, qDot2, qDot3, qDot4;
      float _2q0, _2q1, _2q2, _2q3, _4q0, _4q1, _4q2, _8q1, _8q2, q0q0,
                  q1q1, q2q2, q3q3;

      // Rate of change of quaternion from gyroscope
      qDot1 = 0.5f * (-q1 * gx - q2 * gy - q3 * gz);
      qDot2 = 0.5f * (q0 * gx + q2 * gz - q3 * gy);
      qDot3 = 0.5f * (q0 * gy - q1 * gz + q3 * gx);
      qDot4 = 0.5f * (q0 * gz + q1 * gy - q2 * gx);

      // Compute feedback only if accelerometer measurement valid (avoids NaN in
accelerometer normalisation)
      if (!((ax == 0.0f) && (ay == 0.0f) && (az == 0.0f))) {
```

```
            // Normalise accelerometer measurement
            recipNorm = invSqrt(ax * ax + ay * ay + az * az);
            ax *= recipNorm;
            ay *= recipNorm;
            az *= recipNorm;

            // Auxiliary variables to avoid repeated arithmetic
            _2q0 = 2.0f * q0;
            _2q1 = 2.0f * q1;
            _2q2 = 2.0f * q2;
            _2q3 = 2.0f * q3;
            _4q0 = 4.0f * q0;
            _4q1 = 4.0f * q1;
            _4q2 = 4.0f * q2;
            _8q1 = 8.0f * q1;
            _8q2 = 8.0f * q2;
            q0q0 = q0 * q0;
            q1q1 = q1 * q1;
            q2q2 = q2 * q2;
            q3q3 = q3 * q3;

            // Gradient decent algorithm corrective step
            s0 = _4q0 * q2q2 + _2q2 * ax + _4q0 * q1q1 - _2q1 * ay;
            s1 = _4q1 * q3q3 - _2q3  * ax + 4.0f * q0q0 * q1 - _2q0 * ay - _4q1
                          + _8q1 * q1q1 + _8q1 * q2q2 + _4q1 * az;
            s2 = 4.0f * q0q0 * q2 + _2q0 * ax + _4q2 * q3q3 - _2q3 * ay - _4q2
                          + _8q2 * q1q1 + _8q2 * q2q2 + _4q2 * az;
            s3 = 4.0f * q1q1 * q3 - _2q1 * ax + 4.0f * q2q2 * q3 - _2q2 * ay;
            recipNorm = invSqrt(s0 * s0 + s1 * s1 + s2 * s2 + s3 * s3); //
normalise step magnitude
            s0 *= recipNorm;
            s1 *= recipNorm;
            s2 *= recipNorm;
            s3 *= recipNorm;

            // Apply feedback step
            qDot1 -= beta * s0;
            qDot2 -= beta * s1;
            qDot3 -= beta * s2;
            qDot4 -= beta * s3;
        }

        // Integrate rate of change of quaternion to yield quaternion
        q0 += qDot1 * (1.0f / SAMPLING_FREQUENCY);
        q1 += qDot2 * (1.0f / SAMPLING_FREQUENCY);
        q2 += qDot3 * (1.0f / SAMPLING_FREQUENCY);
        q3 += qDot4 * (1.0f / SAMPLING_FREQUENCY);

        // Normalise quaternion
        recipNorm = invSqrt(q0 * q0 + q1 * q1 + q2 * q2 + q3 * q3);
        q0 *= recipNorm;
        q1 *= recipNorm;
        q2 *= recipNorm;
        q3 *= recipNorm;
}

//***************************************************************************

float invSqrt(float x)
```

```
{
	return 1.0f / sqrtf(x);
}

void PrintFloatVariable(float Float_Variable, uint8_t New_Line)
{

	int32_t i32IPartX, i32FPartX;

	i32IPartX = (int32_t) Float_Variable;
	i32FPartX = (int32_t) (Float_Variable * 1000.0f);
	i32FPartX = i32FPartX - (i32IPartX * 1000.0f);

	if (i32IPartX <= -1)
	{
		i32FPartX *= -1;

		if (New_Line)
			UARTprintf(";%i.%03i\n", i32IPartX, i32FPartX);
		else
			UARTprintf(";%i.%03i ", i32IPartX, i32FPartX);

	}
	else if (i32FPartX < 0)
	{
		i32FPartX *= -1;
		if (New_Line)
			UARTprintf(";-%i.%03i\n", i32IPartX, i32FPartX);
		else
			UARTprintf(";-%i.%03i ", i32IPartX, i32FPartX);
	}
	else
	{
		if (New_Line)
			UARTprintf(";%i.%03i\n", i32IPartX, i32FPartX);
		else
			UARTprintf(";%i.%03i ", i32IPartX, i32FPartX);

	}

}
```